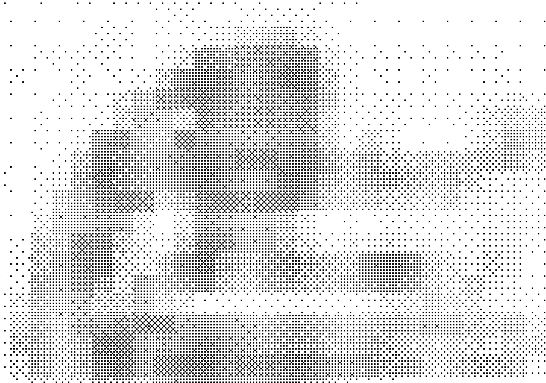




DSP Program Development

DSP程序开发

——MATLAB调试及直接目标代码生成

DSP 程序开发

——MATLAB 调试及直接目标代码生成

李真芳　苏　涛　黄小宇　编著

西安电子科技大学出版社

2003

内 容 简 介

当前，DSP 芯片技术飞速发展，旧型号不断被淘汰，新产品功能越来越强大，而硬件结构和汇编指令也越来越复杂。面对这种形势，DSP 程序开发人员必须转变传统的编程思想，采用开发流程简化的系统级集成环境，以缩短软件开发周期，加快产品的上市时间。

本书针对程序开发人员和 DSP 初学者，介绍了当前最为流行的几种高性能通用 DSP，包括 TI 公司的 TMS320C5000/C6000 DSP 和 AD 公司的 SHARC DSP；详细介绍了当前最新的开发环境及最新的编程思路；介绍了在 MATLAB 及开发工具提供的系统级集成环境下，完成设计、仿真、目标代码产生、调试和运行的过程。

本书面向通信、雷达和电子工程领域的 DSP 算法研究和程序开发人员以及相关专业的研究生和高年级本科生，亦可作为 DSP 爱好者的自学教程。

图书在版编目（**CIP**）数据

DSP 程序开发：MATLAB 调试及直接目标代码生成 / 李真芳等编著.

—西安：西安电子科技大学出版社，2003.10

ISBN 7-5606-1298-9

Ⅰ. D…　Ⅱ. 李…　Ⅲ. 数字信号—信号处理—程序设计　Ⅳ. TN911.72

中国版本图书馆 **CIP** 数据核字（2003）第 083821 号

策　　划　陈宇光　戚文艳

责任编辑　阎　彬

出版发行　西安电子科技大学出版社（西安市太白南路 2 号）

电　　话　(029)8242885　8201467　　邮　编　710071

http: //www.xduph.com　　E-mail: xdupfxb@pub.xaonline.com

经　　销　新华书店

印　　刷　西安文化彩印厂

版　　次　2003 年 10 月第 1 版　2003 年 10 月第 1 次印刷

开　　本　787 毫米×1092 毫米　1/16　印张 24.625

字　　数　587 千字

印　　数　1～4000 册

定　　价　35.00 元

ISBN 7-5606-1298-9 / TN・0239

XDUP 1569001－1

前　　言

如同当前PC机的更新速度一样，DSP技术也在飞速发展，旧型号不断被淘汰，新型号不断产生，且硬件结构和汇编指令越来越复杂。面对这种形势，人们开发DSP产品的开发思路也逐渐发生转变，很多设计者趋向于购买专业公司提供的现成的DSP板或处理机。这样做虽然费用较高，但换来了可靠性，更重要的是节省了硬件的开发周期。因此，现在整个新产品的开发越来越集中在软件开发上。

传统的开发工具利用DSP汇编语言进行低层次的设计，现在这已不适应IT市场的激烈竞争。为了在竞争中获胜，软件开发人员需要利用简单易学的高层次集成环境，来帮助他们摆脱底层设计的困扰，集中精力探索新的算法，获得技术上的突破。作者编写本书的主要目的之一，就是要把一些最新的编程方法、编程思路和调试方法介绍给读者，以帮助他们快速适应不断变化的技术潮流和市场节奏。

MATLAB作为目前最强大的数值计算和分析工具，已被算法研究人员、工程技术人员广泛应用。MATLAB用于DSP的算法模拟/仿真已大有所为，而人们期望着把DSP的所有开发工具都集成在MATLAB中，即在MATLAB统一环境下就可以完成DSP程序开发的整个过程。顺应此潮流，TI公司和SDL公司与MathWorks公司合作，分别针对TMS320C5000/C6000 DSP和SHARC DSP，推出了MATLAB－DSP系统级集成环境，即在MATLAB统一环境下完成概念设计、模拟/仿真、目标代码产生、运行和调试。利用系统级开发环境可以极大地节省耗费在编程和修正错误上的时间，把开发者解放出来，去紧跟前沿科技，探索新的思路，准时提交第一流的产品设计。

编写本书的另一个目的就是为DSP初学者提供一个完整、系统、详细的介绍，这些介绍包括DSP的内部功能结构、源代码编写和开发工具等。学习完本书后，读者基本上就能够进行DSP程序开发了。

本书的内容安排如下：

第1章介绍当前DSP软件设计的过程和可选择的设计方法，介绍了MATLAB在DSP算法模拟、代码调试中的重要作用，也指出了仅用MATLAB去辅助DSP设计所存在的困难。

第2章介绍当前最为流行的几种高性能通用DSP芯片，包括TI公司的TMS320C5000/C6000 DSP和AD公司的SHARC DSP。主要针对程序开发人

员和初学者详细介绍这些DSP的片内CPU核、寄存器、存储器组织、中断、DMA数据传送、几个常用的片内外设以及这些DSP的源代码编写，包括汇编指令、C/C++环境以及C/C++程序和汇编程序接口时注意的问题等。

第3章和第4章介绍两大主流DSP开发工具，其中：第3章详细介绍TMS320C5000/C6000 DSP的开发工具CCS；第4章详细介绍SHARC DSP（ADSP2106x和ADSP2116x）的开发工具VisualDSP++。

第5章、第6章和第7章介绍DSP软件开发的MATLAB系统级集成环境，其中：第5章介绍MATLAB为TMS320系列DSP代码调试提供的支持；第6章介绍MATLAB为TMS320系列DSP的整个代码开发过程提供的支持，在此环境下可以直接从Simulink模型生成TMS320C6000 DSP的可执行代码；第7章介绍如何直接由Simulink模型生成SHARC DSP的可执行代码。

对于熟悉DSP程序开发的人员，可以根据需要，直接选读本书的第1章、第5章、第6章或第7章以及第3章和第4章中的MATLAB应用例子；对于DSP初学者，则需要先仔细阅读第2章、第3章或第4章，然后再阅读其它各章。

每章的最后都提供了一系列思考题，这些思考题可以帮助读者加深对本章内容的理解并开拓读者的思路。

本书第1章由苏涛编写；第2、3章由李真芳、苏涛合写；其余部分由李真芳、黄小宇合写。全书由李真芳统稿。

由于作者掌握的资料和水平有限，加之时间仓促，书中错误之处在所难免，敬请读者批评指正。

作　者

2003年7月

目　　录

第 1 章　MATLAB 与 DSP 软件设计方法综述

1.1　DSP 程序开发过程的演变

1.1.1　DSP 技术综述

数字信号处理技术在最近 20 年获得了广泛的应用。数字信号处理理论和算法是这项技术的一个核心，数字信号处理器(DSP)是这项技术的另一个核心，其中可编程的 DSP 可以将性能很好的信号处理算法方便地应用到实时信号处理中。

MATLAB 是一个强大的分析、计算和可视化工具，特别适用于数字信号处理算法的分析和模拟，使用非常方便。但由于 MATLAB 程序的执行速度相对于实时信号处理来说，仍显得太慢，而 MATLAB 所依赖的平台是计算机等设备，这类设备的体积、功耗不适合于实时信号处理，设备的结构也无法满足实时信号处理所要求的高速数据输入/输出，因此 MATLAB 在数字信号处理技术中，适合于对算法的模拟及对实测数据的事后处理。不过，目前有一种新的技术，可以将 DSP 和 MATLAB 两者密切结合起来，充分利用两者的特长，有力地促进数字信号处理算法的实现。

我们将可编程的 DSP 称为通用 DSP，以区别于 ASIC、CPLD/FPGA 等用硬件进行数字信号处理的器件。本书所讨论的 DSP 都是通用 DSP。就软件可编程而言，DSP 与单片机、PC 机的 CPU 的编程设计方法有类似之处，但 DSP 比单片机的运算速度高得多，又比 CPU 的功耗、设计复杂度低得多。因此，DSP 是进行实时数字信号处理的最佳选择。

为了能低成本、低功耗地进行实时信号处理，各类 DSP 都具备如下特点：

• 采用了数据总线和程序总线分离的哈佛结构及改进的哈佛结构，比传统处理器的冯·诺依曼结构有更高的指令执行速度和数据输入/输出速度。

• 采用流水技术，即每条指令都由片内多个功能单元分别完成取指、译码、取数、执行等多个步骤，从而在不提高时钟频率的条件下减少每条指令的执行时间。

• 具有指令流控制，可以完成无附加开销的循环功能以及延迟跳转指令。

• 具有专门的指令集和较长的指令字，一个指令字同时控制片内多个功能单元的操作。

• 配有独立的算术逻辑单元、乘法器、移位器，可在单周期内完成多次乘、加、移位运算。

• 片内有大容量、多端口存储器，访问速度很快。

• 片内有多条总线可以同时进行取指令和多个数据存取操作。

• 用于寻址的地址寄存器能在其它操作进行的同时，以多种方式自动修改地址寄存器内容，以指向下一个要访问的数据地址。特殊寻址方式有循环寻址、位反序寻址。

• 具有软、硬件等待功能，能与各种类型、不同速度的存储器接口，片内集成的同步 DRAM(SDRAM)控制器支持对高速、大容量存储器的访问。

• 带有 DMA 控制器以及串行通信口等，配合片内多总线结构，使数据块传送速度大大提高。

• 配有中断处理器和定时控制器，可以很方便地构成一个小规模系统或单片系统，易于小型化设计。

• 功耗低，一般为 0.5～4 W，采用低功耗技术的 DSP 只有 0.1 W，待机功耗更低，可用电池供电，对嵌入式系统很适合。因为 DSP 需要的外围设备很少，所以一个紧凑的 DSP 系统的功耗也很低。

采用 DSP 是为了进行高速的实时信号处理，针对各种各样的实际应用，有多种类型、多种档次的 DSP。可以按 DSP 的数据格式，把通用 DSP 划分为定点 DSP 和浮点 DSP。衡量 DSP 性能的主要指标是它完成相应处理任务的速度以及处理精度(数据字长)，最常用的指标是每秒百万次指令执行个数(MIPS，即指令执行时钟)。对大多数定点 DSP 来说，单周期内可以完成一次乘法和一次加法；对浮点 DSP 来说，单周期内可以完成多次乘法和加法。因此，每秒百万次浮点运算(MFLOPS)也是衡量浮点 DSP 运算能力的重要指标。MFLOPS 指标是 MIPS 指标的若干倍。DSP 片内除了运算单元外还有许多其它功能部件，合理设计后，这些功能部件可以同时工作，与此对应的每秒百万次操作(MOPS)也是衡量 DSP 并行操作能力的又一指标，这一指标是 MIPS 指标的若干倍。这些指标都是在最优设计情况下得到的，并不能与 DSP 的实际处理速度等同，因此执行 FFT、FIR 滤波等算法的执行时间就成为一个比较客观的评价标准。在实际设计中，能否达到上述指标，还取决于应用要求、硬件配置、软件编程方法。只有精心设计，才能尽可能地充分利用 DSP 的各种资源，达到比较高的 DSP 运行效率，但这种设计方法往往与设计的通用化、可移植性相矛盾。例如，采用 C 语言设计 DSP 的难度低，设计、调试周期短，设计的 DSP 程序可以在重编译后，运行在各种 DSP 上；而根据特定的 DSP 资源配置，采用特定 DSP 汇编指令编写 DSP 代码的难度大、代码可移植性差，但运行速度比 C 语言程序高得多，常常达到 C 语言程序的 10 倍以上。

1.1.2 DSP 设计过程

利用 DSP 实现一个实时信号处理系统的一般步骤包括：

第一步：根据要求，确定信号处理方案和算法，需要对算法进行原理或功能级的模拟。其中，在满足处理性能的前提下，要对算法的可行性、系统的成本进行评估。

第二步：根据算法，选择合适的实现方法，具体内容就是选择一种合适的 DSP 以及外围器件。这时候的主要工作就是针对这种 DSP 的硬件配置、工作特点，进行算法模拟，考核所选算法在特定 DSP 上能否达到所要求的处理性能和处理速度。

第三步：一方面设计 DSP 硬件电路板；另一方面编写 DSP 程序代码，通常用 C 语言和 DSP 的汇编语言。

第四步：在 DSP 上调试所编写的程序，做到程序和硬件电路都能满足要求。

第五步：把调试成功的 DSP 代码固化到 DSP 目标板上。

然而，并不是所有的 DSP 系统设计都是这一种固定模式。例如，用户可以根据应用需要，购买和采用现成的 DSP 电路板，省去步骤 3 中的硬件设计，用户所作的工作纯粹是编写、调试 DSP 程序。本书的内容不涉及硬件设计，而是集中介绍在 DSP 上实现实时信号处理算法的整个软件设计过程。

实际上，随着 DSP 处理性能的飞速提高，以及用户要求产品的研制周期越来越短，DSP 的设计内容越来越侧重于软件方面。一方面，强大的通用化硬件平台为实现实时信号处理的软件化提供了性能保障，使许多 DSP 设计人员摆脱了硬件设计、配置的困扰，同时也帮助许多纯算法研究人员能轻松地进入 DSP 设计领域。另一方面，DSP 的开发设计环境更加完善，即使要调试 DSP 程序代码，也可以脱离 DSP 硬件电路板。DSP 的调试手段有两种：一种是脱离 DSP 硬件电路板，利用 PC 机的资源模仿 DSP 及其外围电路的工作方式，营造出一个模拟环境，在此环境下调试 DSP 代码，我们称之为软件模拟器(Simulator)的方式；另一种是将 PC 机和 DSP 电路板(称目标板)用专用的 DSP 仿真器及电缆连接起来，从 PC 机上实时监视、控制 DSP 的运行，我们称这种手段为实时仿真器(Emulator)的方式。

模拟器(Simulator)一般用于 DSP 代码的前期调试，只是验证代码的功能和性能。它实用方便，无需添加设备，但模拟器的缺点是速度太慢。例如一段图像压缩代码在真正的 DSP 上运行的时间是 1 秒，而在 2.0 GHz 奔腾 IV 机型上的模拟器下运行时则需要 1 个半小时，其速度相差 5400 倍，因此，用模拟器验证运算复杂度高、运算量大的代码很不合适。此外，用模拟器很难验证 DSP 在实际运行过程中的输入/输出操作。

仿真器(Emulator)对实际的 DSP 硬件目标板进行监控，可以比较真实地得到 DSP 实际运行过程中的状态信息。

1.1.3　DSP 软件设计方法的变革

随着计算机技术、DSP 技术的发展，以 DSP 为核心进行信号处理所用到的软件实现方法经过了多次变革。表 1.1 按 DSP 软件设计方法的变革历程，列出了各种软件设计实现方法及其特点。

表 1.1　不同的软件设计方法

软件设计方法	第一、二步 模拟	第三、四步 编程、调试	开发 周期	难度	代码 效率	代码 长度	可移 植性
① 完全汇编	汇编	汇编	最长	最大	最高	最小	无
② C+汇编	C	汇编	长	最大	最高	最小	无
③ C+C/汇编	C	C/汇编混合	中	中	高	中	部分
④ C+C	C	C	短	小	低	大	完全
⑤ MATLAB+C/汇	MATLAB	C/汇编混合	短	中	高	中	部分
⑥ MATLAB+C	MATLAB	C	最短	小	低	大	完全
⑦ MATLAB 集成	MATLAB	MATLAB	最短	最小	最低	最大	完全

表 1.1 中，代码的效率指运行速度，而代码长度指的是代码所占用的存储器空间。代码的效率和代码长度在大多数情况下是一致的，即代码长度越短，代码的效率/执行速度就越高，但在某些常见情况下，两者是相矛盾的。例如，编写一段循环执行的代码，代码长度

很短，但由于每次循环执行时，DSP 都要判断循环计数器并使用跳转指令(打断了流水线)，执行速度并不高，而如果将这个循环体全部展开，循环体内的代码按循环次数重复写出，代码长度大大加长，但 DSP 不再判断循环计数器，也不使用跳转指令(不打断流水线)，执行速度会显著提高。类似地还有使用子程序还是使用宏的选择问题。对此，将 C 语言等高级语言编写的程序编译成 DSP 的汇编代码时，可以进行编译时的优化选择：使用代码长度的优化，还是代码速度的优化。

表 1.1 中，代码的可移植性指的是，同样的程序能否在不同的 DSP 型号，或同一 DSP 但不同硬件配置的电路板上运行。显然，用 DSP 汇编语言编写的程序可移植性差，而用 C 语言编写的程序可移植性好。

1．汇编语言编程

20 世纪 80 年代，DSP 刚刚出现并应用于信号处理领域，DSP 的性能指标比较低，运算速度大约在 2 千万次每秒，设计人员为达到足够的运算速度，只能采用表 1.1 中的方法①和②，用 DSP 的汇编语言编写高效、专用的程序代码；只是在算法模拟阶段采用 C 语言。

2．C 语言编程

20 世纪 90 年代前半期，DSP 的运算速度接近 1 亿次每秒，设计人员开始采用 C 语言的编程方法，以求降低开发难度、缩短开发周期，但受限于 DSP 的速度、存储器容量、整个硬件系统的成本，只能部分地采用 C 语言设计程序，关键程序段仍然要结合 DSP 的硬件特点，编写 DSP 汇编程序。

目前，最快的 DSP 运算速度已经超过 10 亿次每秒，外围器件，特别是高速、大容量的 SDRAM 型存储器的性能也越来越高。随着 DSP 速度不断提高和硬件成本不断降低，以及用户要求的产品开发周期的不断缩短，C 语言设计的优越性越来越明显，它不仅降低了开发难度，缩短了开发周期，而且有很好的通用性和可移植性，即使淘汰或更换了 DSP 型号，大部分源程序仍然可以移植到新的 DSP 电路板上；同时，DSP、存储器的性能/价格比的大幅提高也有利于克服 C 语言设计的种种局限性，例如代码效率低、代码占用的存储容量太大等缺陷已不再是设计者考虑的主要问题，设计者考虑的主要问题是如何缩短开发周期，尽快推出产品。

3．MATLAB 辅助设计方法

20 世纪 90 年代后期，MATLAB 作为一种有效的信号处理工具出现后，逐渐渗透到 DSP 的设计当中。MATLAB 是一个强大的分析、计算和可视化工具，使用非常方便。用 C 语言要比用汇编语言编程方便得多，而用 MATLAB 又比用 C 语言编程方便得多。

在设计一个实时 DSP 系统前，常常用 MATLAB 对算法在 DSP 上运行的性能进行模拟，以验证算法本身的正确性。这种模拟分为精确的模拟和功能的模拟，这两种模拟都不简单，前者更复杂。在本章后半部分将对此进行介绍。

在 DSP 系统的程序调试过程中，利用 MATLAB 还可以产生模拟数据,供调试 DSP 时使用，并且将 DSP 的处理结果和 MATLAB 的处理结果进行比较和验证。

截止目前，我们在将一个新的数字信号处理算法应用于实际前，最方便的方法就是先用 MATLAB 进行模拟验证，当模拟结果满意时再把算法修改成 C 或 DSP 汇编语言，在目标 DSP 上实现。目标 DSP 可以是实际的一个 DSP 硬件电路板，也可以是 PC 机上的 DSP

软件模拟器。在 DSP 软件设计的整个过程中，这是最花费时间的部分，编程者要花费大量的时间编写程序，并在目标 DSP 上调试程序。当 MATLAB 模拟结果令人满意后，我们常常把 MATLAB 的结果作为标准值，与用 C 或 DSP 汇编语言编写的 DSP 代码执行结果进行比较。这两者之间通常会有差别，出现差别的主要原因在三个方面：① 代码编写有误；② 实时处理时，DSP 外围硬件接口的问题；③ 算法用 DSP 实现时，数据要量化，存在量化误差。考虑到 MATLAB 与 DSP 上的 C 或汇编在运算精度、动态范围上的不同，即使 DSP 上的 C 或汇编程序编写无误，也有必要在目标 DSP 上重新验证算法的性能。在大多数情况下，MATLAB 的模拟结果和 DSP 的运行结果在性能上的差别是很小的，可以忽略，但有时必须留心这一差别。

编写 DSP 的 C、汇编语言程序与编写 MATLAB 程序当然不是一个概念，前者复杂得多，但借助于 MATLAB，可以降低这一复杂度。我们在设计 DSP 软件时的主要工作，就是保证运行在 DSP 上的 C、汇编程序编写正确、无误，一种简便方法就是通过 DSP 的开发工具把目标 DSP 程序运行的中间结果保存到 PC 机的硬盘上，然后再调入到 MATLAB 工作空间中，与 MATLAB 模拟算法的中间结果进行比较，以发现 DSP 程序编写的错误以及由精度问题导致的结果偏差；或者反过来，用 MATLAB 产生模拟数据文件，以供测试 DSP 程序用。此模拟数据文件可以直接包括到 DSP 的程序代码中，也可以通过 DSP 的开发工具把模拟数据文件调入目标 DSP 中，由 DSP 处理，观察或保存其结果，并与 MATLAB 对同一数据的处理结果进行比较。当信号处理比较复杂时，需要分步验证各个中间结果和最终结果。如果是因为精度问题导致的结果偏差太大，需要用 MATLAB 对算法进行修正，再在 DSP 上用 C、汇编语言编写修正的算法程序。

如此过程反反复复，在 DSP 的开发工具、MATLAB 工作空间之间来回多次切换，仍然显得繁琐、不便。对于熟悉并依赖于 MATLAB 的 DSP 开发人员来说，他们特别期望有一种新的工具能够把 MATLAB 和 DSP 开发工具集成在一起，在 MATLAB 下就能完成 DSP 软件开发的全部过程，而对于熟悉 MATLAB 并开始进行 DSP 设计的初学者，或者利用 MATLAB 专门研究算法并关心其可实现性的算法研究/分析人员来说，更希望有这样的一种工具。

4．MATLAB－DSP 集成设计环境(系统级集成环境)

MATLAB 使用方便的一个原因是它是一种解释型的语言。但解释型语言的一个缺点是执行速度很慢，另一个缺点是必须在 MATLAB 环境下才能运行，而安装 MATLAB 环境需要几百兆字节以上的硬盘空间和相当大的计算机内存。只有将其编译成可执行的应用程序，才能提高执行速度，并独立于 MATLAB 环境运行，这样生成的代码长度和需要的内存空间都小得多。一般来说，MATLAB 程序总是先被翻译成 C/C++，然后被诸如 MSC++等开发工具编译成可执行文件。

既然 DSP 可以用 C 语言设计，而 MATLAB 又可翻译成 C 语言，MATLAB－DSP 设计人员、算法研究人员的这一梦想——把 MATLAB 和 DSP 开发工具集成在一起，就顺理成章地变为现实。

在较新版本的 MATLAB(6.0 以上)环境下提供了这一工具，能将 MATLAB 程序转换成 DSP 代码。其方法是：MATLAB 程序先被转换为 C 程序，再针对特定的 DSP 型号、DSP 目标板，编译(转换)成 DSP 汇编指令，最后生成 DSP 的可执行代码。但研究设计人员可以

不去关心MATLAB程序如何转换为C程序，C程序如何转换为特定的DSP汇编指令等这些转换步骤是如何具体实现的，因为这是由MATLAB自动完成的。特别是对于专门研究算法的人员，他们无需熟悉，甚至无需了解具体的DSP硬件结构、功能、指令、DSP的存储器配置，只要在MATLAB环境下，就可检验算法在一种或几种DSP上的实际运行效果。这样，就把在MATLAB下模拟DSP实现某个算法的繁琐过程，以及用C和汇编语言编写、调试DSP代码的复杂性掩盖起来了。用户只要会使用MATLAB，即可在DSP上测试算法。

当然，通过这种方法得到的DSP代码，效率会低得多，这里的效率主要指程序的代码长度、运行速度。因为我们知道，由MATLAB得到的C程序比直接用C语言编写的程序效率低，用C程序编译后得到的汇编代码比直接用汇编语言编写的手工汇编代码效率又要低很多，代码也很长。如果不根据DSP结构和DSP目标板存储器配置对程序代码进行优化的话，生成的DSP代码肯定是低效率的。这种代码很有可能只是可运行、可模拟/仿真的，只具有分析意义。除非DSP的速度相对于算法所要求的高得多，例如采用目前最快的每秒运算15亿次的DSP，即使只有5%～10%的低效率，也能满足需要，否则代码必须优化，才具备实用性。

要对代码进行优化，除了前文讲述的软件层次的编译优化方法外，设计人员还必须熟悉DSP内部的结构特点，并花费较多的时间优化代码和硬件资源间的配置，同时还要考虑到MATLAB程序所生成的DSP代码长度可能远远超出了DSP目标板的总存储器容量。这些工作都是比较繁琐的，对集中精力研究算法的人员，确实不是一件容易的工作。因此，要想在MATLAB环境下高质量地完成DSP软件开发的全部过程，仍有许多工作要等待相关的DSP厂商、MATLAB系列的软件开发商去完善。

1.1.4　MATLAB - DSP集成设计环境下的工具包

目前，MathWorks公司和TI公司联合开发的工具包——MATLAB Link for CCS Development Tools，已经能把MATLAB和TI的DSP集成开发环境CCS(Code Composer Stadiu)及目标DSP连接起来。MATLAB Link for CCS Development Tools作为MATLAB的一个新工具箱被集成在MATLAB 6.5(Release13)以及更新的版本中。利用此工具可以像操作MATLAB变量一样来操作TI DSP的存储器或寄存器，即整个目标DSP对于MATLAB像透明的一样，开发人员在MATLAB环境下，就可以完成对CCS的操作，对DSP目标程序中的函数的操作，可读写DSP中某一段存储器或寄存器，利用RTDX进行实时数据交换等，所有这一切操作只利用MATLAB命令和对象来实现，简单、方便、快捷。MATLAB Link for CCS Development Tools可以支持CCS能够识别的任何目标板，包括TI公司的DSK、EVM板和用户自己开发的目标DSP(C2000™，C5000™，C6000™)板。此工具只用于DSP程序的调试过程，如果把此工具与MathWorks公司和TI公司联合开发的另一工具包——Embedded Target for the TI TMS320C6000™ DSP Platform配合使用，则可直接由MATLAB的Simulink模型生成TIC6000DSP的可执行代码，即在集成的、统一的MATLAB环境下可完成DSP开发的整个过程。

针对ADI公司的SHARC浮点型DSP，也有类似的MATLAB工具包，具备类似的功能。

此外，因为定点DSP的定点型运算和普通的MATLAB模拟中用到的双精度浮点型运

算有很大差别，用 MATLAB 模拟定点 DSP 的运算难度很大，鉴于这一点，MATLAB 还提供了一个针对定点 DSP 的工具包——Fixed-point Blockset，它大大简化了对定点 DSP 的模拟。

本书主要介绍两类与 MATLAB 相关的 DSP 软件设计方法。一类是 MATLAB 辅助下的 DSP 软件设计方法，如表 1.1 中的方法⑤、⑥，把普通的 MATLAB 工具和 DSP 语言设计结合起来的组合方法，即先借助 MATLAB 工具进行算法模拟，再编写 DSP 程序，在 DSP 程序调试阶段用 MATLAB 辅助调试和验证。本章的后续内容将概要介绍这一方法，在第 3、4、5 章中也分别对这一方法进行了介绍。另一类是 MATLAB 环境下的 MATLAB - DSP 软件集成设计方法，即表 1.1 中的方法⑦——完全的 MATLAB 设计方法。本书将在第 6、7 章再对这一方法进行详细介绍。这两种方法各有优缺点，各有适用范围。

1.2　MATLAB 辅助下的 DSP 软件设计

MATLAB 作为一种有效的信号处理工具，已渗透到 DSP 的设计当中。我们在将一个新的数字信号处理算法应用于实际前，将先用 MATLAB 进行模拟验证，当模拟结果满意时再把算法修改成 C 或 DSP 汇编语言在目标 DSP 上实现，并验证。其具体步骤是：

(1) 用 MATLAB 模拟验证算法；

(2) 根据 MATLAB 程序，编写用于 DSP 的 C 或 DSP 汇编语言程序，生成可执行代码；

(3) 在 DSP 开发系统的模拟/仿真工具中，调试并验证代码的正确性、精度和实时性。

对于一个经过 MATLAB 模拟的算法，用 C/汇编语言编写和调试 DSP 程序时有诸多要注意的方面。一个最基本点就是 MATLAB 的数据格式与 DSP 的数据格式有明显差别，特别是与定点 DSP 的差别很大。

MATLAB 本身面对的是科学计算和分析，其数据格式默认为双精度浮点，其精度比 DSP 高得多，动态范围比 DSP 大得多。为了使经过 MATLAB 模拟的算法能够适用于 DSP，在 MATLAB 模拟这一算法时，必须注意使 MATLAB 尽量真实的模拟 DSP 的实际运算过程，这样就必须对普通的 MATLAB 程序进行改进。以下就浮点 DSP 和定点 DSP 设计分别讨论。

浮点数据的动态范围比定点数据大得多。IEEE754 标准的 32 位浮点数据表示范围为 $-1.7014\times10^{38}\sim1.7014\times10^{38}$，最小数据单位(精度)为 $1.175\,49\times10^{-38}$，动态范围为 1536 dB。在编程时几乎可以不考虑浮点数据的溢出，其处理精度也比定点方式高得多，因此编写浮点 DSP 处理程序(无论是用 DSP 汇编指令还是用高级语言)比编写定点 DSP 处理程序要简单方便得多。例如用浮点 DSP 进行浮点数据的 FFT 变换，程序简洁、精度高，而定点 DSP 的 FFT 程序要复杂得多。此外，浮点 DSP 同时可以进行 32 位或 24 位的定点数据运算，这也比 16 位的定点 DSP 强；在进行函数运算时，浮点 DSP 的效率也高得多，例如以迭代方式进行除法运算或求平方根时，浮点 DSP 的指令数就少得多。应注意的是大多数浮点 DSP 具有 40 位的扩展精度寄存器，其表示的 40 位浮点数比 IEEE 标准的 32 位浮点数增加了 8 位尾数，因而这些运算寄存器的运算精度也较高。

但定点 DSP 的优势是结构简单，因而在速度、成本、功耗上均强于浮点 DSP。不过，本书不详细讨论究竟应采用定点 DSP 还是浮点 DSP 来实现一个算法的问题。

无论是浮点 DSP 还是定点 DSP 都可进行更高精度的数学运算，例如 16 位定点 DSP 可模仿进行 32 位/64 位定点运算和浮点运算，而 32 位浮点 DSP 可进行 64 位双精度浮点运算。但这两种做法的代价是每次运算都要用许多条指令，使程序的执行效率和可读性变差。

在设计 DSP 程序前和 DSP 程序调试过程中，经常要借助于 C、MATLAB 等工具模拟和验证 DSP 的处理效果和处理过程。对浮点 DSP 来说，这些步骤要简单得多。在 PC 机上，若利用 C 语言的 32 位单精度浮点格式来模拟验证，PC 机和 DSP 两者的执行结果只会有细微的差别，因为 DSP 有 40 位的扩展精度寄存器。在 PC 机上，若用 C 或 MATLAB 的双精度浮点格式来模拟验证时，会得到比 DSP 更精确的处理结果，但两者的差别仍很小，相对误差一般不超过 10^{-6}。

用高级语言模拟和验证定点 DSP 的处理过程就很复杂，一般不能直接使用 MATLAB。用 C 语言编程时必须做到数据格式、中间结果、溢出控制、移位等与 DSP 的操作相一致，这就过于费时费力。一种粗略的替代方法是用 MATLAB 或 C 只作原理功能的模拟和验证，这就可以采用浮点处理方式，然后再编写实际的 DSP C 语言程序或汇编代码。若 DSP 程序正确，两种结果应该是接近的。若由 DSP 程序得不到原理模拟结果，则应分析多种原因，除了 DSP 程序编程错误外，其它原因还有精度不够、中间过程发生溢出或饱和等。

1.2.1 MATLAB 模拟浮点 DSP

IEEE754 标准的单精度格式中，数据的 32 位从高到低是这样规定的：1 个符号位，8 个指数位，23 个尾数位；双精度(64 位浮点)格式是这样规定的：1 个符号位，11 个指数位，52 个尾数位。

双精度格式比单精度格式所表示的数值范围和精度都高很多，这对复杂的科学计算是很有意义的。而对实时信号处理来说，在多数情况下，不需要有像科学计算那样高的精度，甚至可以讲，双精度格式和单精度格式进行处理所得到的结果几乎无差别。这在用 MATLAB 模拟的许多 DSP 设计中已经被验证。因此，对于浮点 DSP 来说，多数情况下，DSP 软件设计步骤的第一步和第二步都比较简单，因为 DSP 的 32 位/40 位浮点数据格式的处理精度仅稍差于 MATLAB 的双精度(64 位浮点)格式。例如求解 10 阶非奇异方程，两者的精度仅相差约 10^{-6}，可以忽略。但在少数情况下，MATLAB 的模拟效果并不能直接由 DSP 实现。下面是两个例子。

例如求解一个高阶奇异方程，因为 MATLAB 的精度高、动态范围大，可以得到正确解，而 DSP 则不能得到正确解。对此，有两种解决方法：一种是用 DSP 进行双精度数据格式的运算，但 DSP 的处理速度会大大降低；第二种是修改算法，增加运算冗余量，例如采用对角加噪等技术，虽然能得到正确解，但处理误差会有所增加。

再例如设计带通 IIR 滤波器时，若用 MATLAB 设计出满足要求的一组滤波器系数，是双精度格式，其极点在单位圆内，但很接近单位圆。放到 DSP 程序中后，DSP 将系数截取成单精度的 32 位浮点数，其极点位置移到了单位圆外，造成这个 IIR 滤波器不稳定。输入有限的一段数据(例如冲激函数)后，稳定的滤波器的输出幅度会越来越小，不稳定的滤波器的输出幅度会越来越大。如有 MATLAB 语句：

```
n=4;
wn=[0.095 0.105];            // 滤波器的归一化通带，归一化中心频率为 0.1
[bn,an]=butter(n,wn);
```

上述 MATLAB 语句设计了一个有 4 个二阶节的 IIR 滤波器，对采样率的一半归一化，通带为 wn，向量 an、bn 是得到的双精度系数，如表 1.2 所示。

表 1.2　IIR 系数截断前后的变化

双精度系数	截断为 IEEE 单精度格式系数
b1=5.8451391939403638e- 008	b1=5.8451391e- 008
b3=- 2.3380556775761455e- 007	b3=- 2.3380557e- 007
b5=3.5070835163642185e- 007	b5=3.5070835e- 007
b7=- 2.3380556775761455e- 007	b7=- 2.3380557e- 007
b9=5.8451391939403638e- 008	b9=5.8451391e- 008
b2=0, b4=0, b6=0, b8=0	b2=0, b4=0, b6=0, b8=0
a2=7.53130698945884	a2=7.5313067
a3=- 25.188874187641773	a3=- 25.188873
a4=48.833357557129574	a4=48.833355
a5=- 60.001717717869219	a5=- 60.001717
a6=47.841275619750178	a6=47.841274
a7=- 24.175816452602906	a7=- 24.175816
a8=7.0815628434767826	a8=7.0815625
a9=- 0.92118192919123643	a9=- 0.92118192

此组系数的极点在单位圆内，滤波器是稳定的。但将此系数装入浮点 DSP 后，系数被截断为 32 位 IEEE 单精度格式。此时，这组系数的极点移到了单位圆外，滤波器不再稳定。有三种方法可以解决这一问题。第一种，可以采用数字信号处理的滤波器设计方法，修正单位圆半径，使其稍小于单位值 1。据此，在用 MATLAB 设计 IIR 滤波器时，就可以迫使极点远离单位圆，这样，系数、输入数据的舍入误差以及运算过程中的舍入误差，都不会造成滤波器不稳定。此种方法需要的半径换算较为繁琐。第二种，可以在用 MATLAB 设计 IIR 滤波器时，加约束条件，使所有极点远离单位圆，例如约束极点半径小于 0.95，这样，系数截断后，极点仍在单位圆内。第三种，可以采用尝试法，即稍微修改滤波器参数，如修改带宽、中心频率等，得到多组系数，被截断为 32 位浮点数后，用 MATLAB 的滤波器设计分析工具 FDATool 观察其极点位置，从中挑选出一组“看似”稳定的系数。一般来说，若原始系数的极点位置远离单位圆，系数被截断后，极点位置也不会越出单位圆。

如表 1.2 所示，浮点 DSP 在将双精度数据截断为 IEEE 单精度格式时，只保留约 10 个十进制有效位，末位不一定采用四舍五入的截取法。要得到截取后的数据，并在 MATLAB 工具中对其进行分析，有两种方法。一种方法是先把 MATLAB 下得到的双精度数据存入数据文件(在 FDATool 工具中用导出命令 Export)，随后在 DSP 的开发环境中按单精度浮点格式，把此数据文件调入 DSP 片内，数据将自动被截断，然后再将截断后的数据存入数据文件，最后再调入 MATLAB 工作空间(在 FDATool 工具中用导入命令 Import)。这一方法比

较费时，我们可以采用另一种方法，即根据这两种数据格式编写 MATLAB 程序，逼真地模仿 DSP 对双精度数据的截取方式，然后再分析截取效果。

如果试图换一种方法，即在 MATLAB 中也像 DSP 一样用单精度格式处理数据，其难度是很大的(若用 40 位扩展精度格式将更复杂)，我们必须在每次运算后，把双精度的结果截断为单精度格式。即使这样，仍会与 DSP 的处理方式不一样，因为在 MATLAB 运算时用的是 64 位乘法器，而 DSP 并没有诸如 64 位乘法器这样的运算部件。所以，MATLAB 的单精度格式模拟只能是一种近似。而且，许多 MATLAB 函数内部用的是双精度格式，这是我们无法干预的，因为我们无法将这些函数内部的运算近似成单精度格式。如果我们干脆不用这些函数，而是用基本的 MATLAB 语句来编程，就丢失了 MATLAB 的大多数优点。所以，我们一般不用单精度格式去进行 MATLAB 模拟。而在后文中用 MATLAB 模拟定点 DSP 时，定点和浮点处理的差别太大了，我们必须按照定点 DSP 的格式去模拟算法。相对来说，MATLAB 模拟定点 DSP 的运算比较简单。

1.2.2 了解定点 DSP 数据格式

对于定点 DSP 来说，其软件设计步骤的第一步和第二步都要复杂得多，要花费较多时间。这是因为：定点 DSP 的数据格式大都为 16 位(少数是 24 位或 32 位)，其处理精度根本无法和 MATLAB 的双精度格式相比。所以在用 MATLAB 模拟时，所有计算应当采用与 DSP 一致的定点处理方法。

定点数又分为有符号数和无符号数，以下讨论都是针对有符号数的。定点意味着数据中的小数点位置是固定的，当小数点位置在数据末尾，数据就是一个纯整数。对于 16 位定点整数来说，数据的取值范围是-32 768(8000H)～32 767(7FFFH)，这种整数形式又称为 Q0 格式。当小数点位置在符号位后面时，对于 16 位定点数来说，数据的表示范围为 -1～0.999 969 5，每一个量化单位相当于 0.000 030 5(即 1/32 768)，数据是一个纯小数，这种表示形式又称为 Q15 格式。Q0 和 Q15 是最常用的定点格式。除了这两种格式外，小数点位置在第 n 位，数据就是 Qn 格式。DSP 编程人员应始终清楚小数点位置和数据位宽，并在每次乘法、加法运算后，对结果进行适当的处理。不管是哪一种格式，16 位定点数据的动态范围始终是固定的，动态范围是所能表示的取值范围和最小数据单位(量化单位)的比值。动态范围常用对数表示。例如，16 位定点整数所能表示的取值范围是 2^{16}，最小数据单位是 1，动态范围即 $20\lg2^{16}=96$ dB；而 16 位定点小数所能表示的取值范围是 2，最小数据单位是 1/32 768，动态范围也是 $20\lg2^{16}=96$ dB。但对于不同的 Qn 格式，在运算过程中对结果数据的处理方式也不同。

1. 整数的处理

两个 16 位带符号整数进行简单的乘加运算后，结果的有效数据位数可能增加，如果仍用 16 位数据存储结果，有可能出现溢出现象。例如，两个 16 位数据相乘的结果是 32 位，最高两位都是符号位，通常 DSP 只能保存 16 位结果，因为 DSP 的存储器和大部分寄存器都是 16 位的，只有乘法器和累加器是 32 位～40 位的。选取相乘结果的高位可以避免溢出，但相乘结果数值太小时，高位大部分或全部都是符号位，数值的处理精度就会大大降低；选取相乘结果的低位进行保存，可以保证精度不损失，但数据很可能溢出且会得到错误的

结果。如何保证数据不溢出且精度损失最小化，是定点数据处理时必须考虑的问题，也是一个比较复杂的问题。设计者必须预先估算运算过程中输入数据、中间结果、输出数据的取值变化范围，通过左移(放大)、右移(缩小)、乘因子等手段控制数值的范围。在最后一步还应计算出数据的放大/缩小比例因子，必要时将根据此因子，对结果进行修正/还原，以将所得的结果控制在规定的数值范围之内。当然，也可以采用 32 位定点数据格式对运算结果进行保存和后续处理，但这样的话，DSP 程序就太复杂，效率太低了。

估算数值范围可以有多种手段：

(1) 了解处理过程，分步计算出数据的取值范围，这要求输入数据的取值范围、运算处理过程是固定或明确的。

(2) 当输入数据变化范围很大时，整个处理过程很复杂，特别是采用自适应算法时，无法或难以估算数据的取值范围，这时，可以利用 C 或 MATLAB 工具进行模拟，得出各种情况下数据的取值范围。

在一般情况下，可以预先知道数据的数值范围。下面以一个 FIR 滤波器为例作一简单描述，来确定在定点数据的处理时，如何选择合适的位段，以保证不溢出、精度又较高。

输入数据为 12 位补码，且认为是满幅的，即变化范围为$-2^{11}\sim2^{11}-1$，即$-2048\sim2047$。

FIR 滤波器阶数为 20，原始系数已归一化，在$-1\sim+1$之间。为了能用定点 Q0 格式处理，同时考虑到乘法器和累加器只有 32 位，所以给 20 个系数同时都乘以$(2^{15}-1)$，将小数点后的数舍去，转化为 16 位整数。对系数的这种处理是在编写 DSP 程序前，用 C 或 MATLAB 等预先完成的。

当输入数据为最大值，即系数全为 1(放大后为 32 767)时，数据和系数符号相同，乘法累加结果达到最大，为 $2^{11}\times2^{15}\times20<2^{31}$(不考虑符号位)，此时，将乘法累加结果的低 14 位丢掉，保存 30 位结果的高 16 位即可。实际上，系数不全为 1，数据和系数符号也不会相同，所以乘法累加结果不可能达到这么大。因此可以将系数分别取绝对值，相加后再乘以最大的输入数据，就得到(可能的)最大的乘法累加结果。如果这一结果是 26 位，那么，丢掉低 10 位是最优的选择。如果输入数据或 20 个系数的大多数数据的绝对值都很小，保存结果时要从累加器中丢掉高位的若干个多余符号位，选取中间的 16 位。由此，也可知道采用这种 FIR 滤波方法时，处理过程中的放大和缩小因子。

这样，在 FIR 滤波过程中不会造成数据溢出。

各种 DSP 的指令系统中都包括了丰富的移位指令，可以在乘加运算前将输入数据移位后再运算，也可以在运算完成后，先将结果移位再保存到存储器中。要注意的是定点 DSP 的累加器位数较大，通常大于乘法器结果的位数(32 位)，可以从中选择合适的位段放入常规的 16 位存储器中。此外，DSP 还有饱和模式，即当结果溢出时，将其置为正的或负的最大值，这样就避免了出现严重的错误。当溢出幅度不大时，这种模式对运算结果的影响不大，有时是可以接受这种误差的。例如运算结果为 32 769(十六进制为 8001)，不设定饱和时，此值被当成-32 767(十六进制也是 8001)，设定饱和时，此值被限幅为 32 767(十六进制为 7FFF)，误差不大。DSP 还能自动检测和查询溢出及饱和标志，以响应溢出中断，这样可以及时发现溢出并加以解决。

DSP 在进行定点处理时还可以用归一化指令消去多余的符号位。例如两个 16 位数相乘后得到的 32 位结果中，若干高位都是符号位，用归一化指令可以自动地将多余符号位舍去，

这相当于数据左移，数据被放大，从而使精度最高化。

为了在定点运算时保证数据既不溢出，又有足够的精度，可采用一种块浮点方法，即对一批数据，按其中绝对值最大的数对整批数据进行归一化，去掉多余的符号位。该方法是先求出这一批数据中冗余符号位最少的数(对应绝对值最大的数)，例如这个最少的冗余符号位数是 M，然后用归一化指令逐个处理每个数据，使其都左移 M 位。这种做法的根据是：在许多实际应用中，我们并不关心处理结果的绝对大小，只关心一批数据结果的相对幅度。块浮点 FFT 是一个例子。在所有定点类型的 FFT 运算时，为防止数据溢出，在每级蝶形运算后都要对蝶形运算的结果右移(缩小)若干位。块浮点 FFT 方法则根据结果的大小来决定是否移位，蝶形运算结果不大时，就不缩小结果。这样的移位位数是动态的，对小信号、大信号的放大/缩小倍数是不一样的。虽然这样的 FFT 结果的绝对大小不具备可比性，但我们常常只关心 FFT 结果中各分量的相对大小。

2. 小数的处理

16 位小数表示方法如下，从高到低位，各位为 1 时，依次表示的数值是：

$$-1,\ 1/2,\ 1/4,\ 1/8,\ \cdots,\ 1/2^{15}$$

最高位是符号位。将一个 16 位整数除以 32 768 即是其表示的小数，即把一个 16 位数看成整数或小数时，它们相差 32 768 倍。如 6000H 表示 0.75, C000H 表示-0.5。

小数的处理和上文的整数处理方法基本一样，也要防止溢出和精度损失，采用的方法有移位、定标、归一化等。与整数处理方法所不同的是对乘法结果的处理和对一些算法的编程。

两个 16 位整数相乘时，结果会越来越大，即乘积向左增长；而两个小数相乘时，结果会越来越小，即乘积向右增长。当两个 Q15 格式的纯小数相乘时，必须将乘积结果左移 1 位以去掉多余的符号位，才能使结果正确。定点 DSP 在小数相乘方式下，能对乘法结果自动左移 1 位，因此其小数乘法比其整数乘法方便。

某些算法程序对整数和小数格式是不通用的，例如求一个数 x 的平方根 y 时，若把 x、y 当作小数，必然有 $x \leqslant y$；若把 x、y 当成整数，必然有 $x \geqslant y$，因此，如果已经有了一个求 Q15 格式小数的平方根程序，也不能直接用来求整数的平方根，必须对此程序的结果进行修正。因为整数与 Q15 小数格式相差 32 768 倍，所以只要将结果除 sqrt(32 768)即可。但由于 DSP 的除法实现较难，可以先将输入量 x 左移 1 位(乘 2，但不能溢出)，把其当成小数，求平方根；再将结果右移 8 位(除以 sqrt(2×32 768))，即可得到整数 x 的平方根结果。

为了提高处理精度，可以用定点 DSP 来模拟浮点处理，例如采用 16 位浮点格式，但效率很低，精度和动态范围也不及真正的 32 位浮点处理。若模拟 32 位浮点处理，效率更低。这种模拟方法也使程序的可读性变差，有时只在程序的一部分运算中采用。

1.2.3 MATLAB 精确模拟定点 DSP 运算

DSP 的运算都归结为乘法、加法运算。在保存乘法、加法运算的运算结果时，必须考虑在不溢出的前提下，如何提高数据的精度，即保存尽量多的有效位数(不包括符号位：高位连续的 0 表示正号，高位连续的 1 表示负号)。常用的定点 DSP 数据位宽 16 位，乘法器的结果有 32 位，累加器有 32 位或 40 位(如最常用的 TMS320C5000 的累加器是 40 位)。下面举例介绍如何利用 MATLAB，模拟定点 DSP 运算。

1．乘法运算

两个 16 位定点数相乘后，要将其舍去若干位，只保留 16 位。若两个数据是整数(Q0 格式)，MATLAB 的程序是：

```
ic=ia*ib;            % ia,ib 都是 16 位整数
ic=fix(ic/32768);    %为了不溢出，不计多余的符号位，只保存结果的高 16 位，相当于缩小
                     %了 32768 倍
```

当 ia,ib 的绝对值较小时，可以取中间 16 位，例如用 ic=fix(ic/2^{10})只丢弃低 10 位。这样处理后，数据幅度被缩小了 2^{10} 倍。

若两个数据是纯小数(Q15 格式)，因为 MATLAB 只能模拟 DSP 整数运算，所以纯小数必须先转化为整数。MATLAB 的程序是：

```
ia=fix(ia*32768); ib=fix(ib*32768);    % ia,ib 都是 16 位纯小数，先转化为 16 位整数
ic=ia*ib;                              % ia,ib 都是 16 位整数
ic=fix(ic/32768);                      % 丢弃低 15 位
ic=ic/32768;                           % 还原成小数
```

这样处理后，数据幅度不被放大/缩小，但最后结果 ic 的绝对值不能大于 1。

当 ia, ib 的绝对值较小时，可以取小数乘法结果的中间 16 位，例如：

```
ic=fix(ic/16384);                      %只丢弃乘积 ic 的低 14 位
ic=ic/32768;                           %还原成小数
```

这样处理后，数据幅度被放大。

2．加法运算

两个 16 位定点数相加，或者 31 位乘法结果累加后，都可能溢出，为了防止溢出，要将最低位舍去，只保留较高的 16 位。两个 16 位定点数相加的 MATLAB 程序是：

```
ic=ia+ib;                              % ia,ib 都是 16 位整数
ic=fix(ic/2);                          % 为了不溢出，丢弃最低 1 位
```

这样处理后，数据幅度被缩小了 2 倍；当 ia, ib 的绝对值较小时，也可以不做这一步。

若两个数据是纯小数(Q15 格式)，MATLAB 的程序是：

```
ia=fix(ia*32768); ib=fix(ib*32768);    % ia,ib 都是 16 位纯小数,先转化为 16 位整数
ic=ia+ib;                              % ia,ib 都是 16 位整数
ic=fix(ic/2);                          % 丢弃低位
ic=ic/32768;                           % 还原成小数
```

这样处理后，数据幅度被缩小了 2 倍；当 ia, ib 的绝对值较小时，也可以不丢弃最低位。

上述程序还没有考虑数据的大小因素：对大数据，一般保存运算结果的高位；对小数据，若保存运算结果的高位，则精度损失太大，所以一般保存运算结果的低位；对介于大数据和小数据之间的数据，一般应保存运算结果的中间位。这些都很难一概而论，必须结合具体的实际数据，进行多次尝试才行，因此程序就更复杂了。

3．函数运算

DSP 可以很方便地直接进行乘法、加法、减法、移位运算，但对于一些常用的非线性运算就必须采用特殊的方法。常见的非线性运算包括：除法、求平方根、三角函数、反三

角函数、指数、对数等。根据不同类型的运算，结合具体 DSP 的运算功能和配置的存储器容量，可采用不同的运算方式。最常采用的是迭代、级数展开等方法。级数展开法比较规范，在相关数学手册上都能查到每种非线性运算的级数展开公式，但应用时要注意输入量的取值范围。

所有这些算法，在对其性能并不明确之前，直接编写 DSP 汇编代码，可能要做些无用的工作。例如某种算法从原理上讲是可行的，用高精度的数据格式进行模拟也能满足要求，但用定点 DSP 编写 DSP 汇编代码时，却因精度不够而出了问题，这时只能另找其它算法，甚至更换 DSP 型号。

如果事先利用 MATLAB 来模拟 DSP 的具体运算过程，确定某个方法的精度能满足要求后，再编写 DSP 汇编代码，显然能避免走弯路。

虽然从后面的例子中，我们会觉得 MATLAB 的这些模拟程序比较繁琐，但它仍比编写 DSP 汇编指令方便得多；此外，可以利用 MATLAB 丰富的数学、图形工具帮助我们分析、比较各种信号处理算法的性能。这对前期的算法模拟及确定 DSP 处理方法还是很有意义的。

1) 求 sin(x)

三角函数是常用的运算。例如 sin(x)用以下 5 项泰勒级数展开就能达到很高的处理精度：

$$\sin(x)=x-x^3/3!+x^5/5!-x^7/7!+x^9/9!=x(1-(x^2/6(1-x^2/20(1-x^2/42(1-x^2/72)))))$$

x 的单位是弧度，按照上式，1/6，1/20，1/42，1/72 都可以预先算好，放在一个常数因子表中，多次使用的 x^2 也可以在一开始即算出并存起来，这样整个运算仅包括不多的乘法和加减法。

上式是 sin(x)关于 0 点的泰勒级数展开式，当输入量 x 在 0 点附近时，误差较小，随着 x 偏离 0 点，误差会变大。

用 MATLAB 的双精度模拟此级数运算，当$|x|\leqslant p/4$ 时，相对误差为 10^{-6}；当 $p/4<|x|<p/2$ 时，误差为 6×10^{-4}。但定点 DSP 的乘法器结果必须舍去一些位，所以误差要比这个数值大(假定每次乘法结果舍去低 15 位)。此例中的 DSP 采用定点小数格式，但 MATLAB 只能模拟 DSP 整数运算，小数格式和整数相差 32 768 倍。为了用 MATLAB 模拟此例中的 DSP 运算，要将小数乘以 32 768 转化为整数。因此，以上式的 $1-x^2/42(1-x^2/72)$为例，MATLAB 的相应程序应写为：

```
d42=fix(1/42*32768); d72=fix(1/72*32768);
x=fix(x*32768);
x2=fix(x*x/32768);
y=fix(x2*d72/32768);
y=32768- y;
z=fix(x2*d42/32768);
y=fix(y*z/32768);
y=32768- y;
y=y/32768;
```

2) 求平方根

求平方根也能用级数展开法，此法适用于输入数据在 0.5～1 之间的运算。

下例中，对输入 y 先求 x=y-1，再分别用 5 阶、7 阶级数展开，比较其精度。

5 阶结果：

b5=1+x/2-0.5*(x/2)^2+0.5*(x/2)^3-0.625*(x/2)^4+0.875*(x/2)^5;

7 阶结果：

b7=1+x/2-0.5*(x/2)^2+0.5*(x/2)^3-0.625*(x/2)^4
+0.875*(x/2)^5-16/21*(x/2)^6+33/64*(x/2)^7;

以下 MATLAB 程序对均匀分布在 0.4～1 之间的 60 个点，模拟用级数展开法求平方根的精度：

```
N=60;
for i=1:N
  y(i)=0.4+0.6*i/N;
  x=y(i)-1;
     %b5(i)=1+x/2-0.5*(x/2)^2+0.5*(x/2)^3-0.625*(x/2)^4+0.875*(x/2)^5;
  x2=fix(x/2*32768);          %将 x/2 转化为整数或负数，当 y=0.5～1 时，x2=0～-0.25
  x2_2=fix(x2*x2/2);          % 0.5*(x/2)^2，DSP 的乘法器结果为 32 位，而它具有 32 位累加器，
                              % 因此暂不舍去低位
  x2_3=fix(x2*x2/32768);      % DSP 的乘法器输入限定为 16 位，多级乘法时，每次都要舍去低位
  x2_3=fix(x2_3*x2/2);        % 0.5*(x/2)^3, 32 位
  x2_4=fix(x2*x2/32768);      % DSP 的乘法器输入限定为 16 位，多级乘法时，每次都要舍去低位
  x2_4=fix(x2_4* x2_4*5/8);% 0.625*(x/2)^4, 32 位
  x2_5=fix(x2*x2/32768);      % DSP 的乘法器输入限定为 16 位，多级乘法时，每次都要舍去低位
  x2_5=fix(x2_5* x2_5/32768);
  x2_5=fix(x2_5* x2*7/8);     % 0.875*(x/2)^5, 32 位
b5(i)=32768+x2-x2_2+x2_3-x2_4+x2_5;      %DSP 中 32 位累加器的结果
b5(i)= fix(b5(i)/32768);      % 最终结果，与 DSP 结果一致

    %b7(i)=1+x/2-0.5*(x/2)^2+0.5*(x/2)^3-0.625*(x/2)^4+0.875*(x/2)^5-16/21*(x/2)^6
    %          +33/64*(x/2)^7
  x2=fix(x/2*32768);          %将 x/2 转化为整数或负数，当 y=0.5～1 时，x2=0～-0.25
  x2_2=fix(x2*x2/2);          % 0.5*(x/2)^2，DSP 的乘法器结果为 32 位，而它具有 32 位累加器，
                              % 因此暂不舍去低位
  x2_3=fix(x2*x2/32768);      % DSP 的乘法器输入限定为 16 位，多级乘法时，每次都要舍去低位
  x2_3=fix(x2_3*x2/2);        % 0.5*(x/2)^3, 32 位
  x2_4=fix(x2*x2/32768);      % DSP 的乘法器输入限定为 16 位，多级乘法时，每次都要舍去低位
  x2_4=fix(x2_4* x2_4*5/8);         % 0.625*(x/2)^4, 32 位
  x2_5=fix(x2*x2/32768);      % DSP 的乘法器输入限定为 16 位，多级乘法时，每次都要舍去低位
  x2_5=fix(x2_5* x2_5/32768);
  x2_5=fix(x2_5* x2*7/8);           % 0.875*(x/2)^5, 32 位
  x2_6=fix(x2*x2/32768);            % DSP 的乘法器输入限定为 16 位，多级乘法时，每次都要舍
                                    % 去低位
```

```
    x2_6=fix(x2_6* x2/32768);        % (x/2)^3
    x2_6=fix(x2_6* x2_6*16/21);      % 16/21*(x/2)^6, 32 位

    x2_7=fix(x2*x2/32768);           % DSP 的乘法器输入限定为 16 位，多级乘法时，每次都要舍
                                     % 去低位
    x2_7=fix(x2_7* x2/32768);        % (x/2)^3
    x2_7=fix(x2_7* x2_7/32768);      % (x/2)^6
    x2_7=fix(x2_7* x2*33/64);        % 33/63*(x/2)^5, 32 位
  b7(i)=32768+x2- x2_2+x2_3- x2_4+x2_5- x2_6+x2_7;    %DSP 中 32 位累加器的结果
  b7(i)= fix(b7(i)/32768);           % 最终结果, 与 DSP 结果一致
  end
  ideal=sqrt(y);                     % 理想结果
  figure(1);
  subplot(2,1,1);
  plot(y, b5, 'b',y,b7, 'k', y, ideal 'r'); zoom on;grid on; title('兰=5 级，黑=7 级，红=理想');
  b5err=(ideal- b5)./ideal;          % 相对误差
  b7err=(ideal- b7)./ideal;
  subplot(2,1,2);
  plot(y,b5err, 'b', y, b7err, 'k'); zoom on;grid on; title('相对误差兰=5 级，黑=7 级');
  ave=sum(abs(b5err))/N; maxerr=max(b5err);     %统计平均误差和最大误差
```

3) 计算反三角函数

反三角函数也是一个常用的非线性运算，它的运算复杂度很高，而且需要根据输入数据的取值范围，采取不同的运算公式。反正切函数是最常用的反三角函数，它涉及对一个复数求相角。但求反正切并不容易。反正切的泰勒级数(关于 0 点)展开式为：

$$arctan(x)=x- x^3/3+x^5/5- x^7/7+ \cdots \quad |x|<=1$$

可见，x 必须在- 1～+1 之间，对应的角度为- p/4～p/4。实际上，当 x 较大时，如|x|接近 1 时，此级数收敛很慢。当|x|>1 时，此级数不收敛，解决办法是用反余切，即令 y=1/x，则|y|<1。反余切的泰勒级数展开式为：

$$arccot(y)= p/2- y^3/3+y^5/5- y^7/7+ \cdots \quad |y|<=1$$

则有：

x>=0, arctan(x)= p/2- arccot(y)；

x<0, arctan(x)= - p/2- arccot(y)。

与前类似，利用 MATLAB 模拟可以发现，反正切和反余切这两个级数展开式的收敛速度都太慢，当|x|接近 1 时，即使利用 21 阶级数:$arctan(x)= x- x^3/3+x^5/5- x^7/7+\cdots+x^{21}/21$，误差也高达 5%。为此，可以转而求(arctan(x))/2;

设 x=tanA，因为 tanA=2*tan(A/2)/(1- tan(A/2)*tan(A/2))，即 tan(A/2)=(- 1+sqrt(1+x*x))/x，用级数展开法求出 A/2，从而得到 A。

当 x=p/4，即 tan(x)=1 时，此方法用 9 阶级数的精度达 $1.8*10^{-5}$。

实际上，对一个复数求相角时，还要用到除法。复数 x+jy 的相角求法是：

当|y|<=|x|时，求 arctan(y/x)；

当|y|>=|x|时，求 p/2- arctan(x/y)。

可以先求出 y/x 或 x/y，再用级数法展开求角度。而利用

$$\arctan(y/x)=p/8+0.730\,32*(y^2-x^2+2*x*y)/(0.86*y^2+1.86x^2+x*y) \qquad x>=y>=0,\ x>0$$

是一个更快捷的方法，可直接从 y、x 求出角度。当角度在- p/8～p/8 之间时，误差为 $5.6*10^{-5}$。

我们知道，级数的项数越多，DSP 的代码就越长，执行就越费时间。因此模拟时，应该确定选择多少阶级数就能满足精度要求，这个精度要求可以仅仅从此函数确定，也可从整个系统的处理要求来确定。如果系统中其它处理环节的精度不高，单纯提高某一环节的精度也是没用的。在后一种情况下，就需要用 MATLAB 模拟整个系统中各个环节的处理过程，如下所述。

4. 模拟 FIR 滤波器

这里以一个相对简单的 FIR 滤波器为例，介绍用 MATLAB 模拟整个系统中的各个环节的处理过程。实际上，MATLAB 中提供了类似的软件工具包，只是它不能正好满足我们特定的要求。

MATLAB 模拟一个 FIR 系统的框架是：

(1) 产生数字 FIR 滤波器的系数，待定的参数有通带、阶数、系数的位数；

(2) 产生模拟信号源；

(3) 按照一定的采样率、采样位数对信号进行量化；

(4) 按照 DSP 的运算方式进行 FIR 滤波，主要是乘法、加法、移位等运算；

(5) 选择合适的位段后，将结果输出。

上述的各个环节对整个系统的处理性能都有影响，各环节间也相互制约。例如，如果信号量化的位数太少，即使 DSP 处理的精度很高，也无法保证性能。

MATLAB 的滤波器设计分析工具 FDATool 可以对量化前后的滤波器进行详尽的分析，后面将作介绍。

可见，用 MATLAB 精确模拟 DSP 算法本身就很费时间。因此，在许多情况下，我们仅用 MATLAB 进行功能模拟，不详细涉及具体的 DSP 乘、加操作，这就简单得多了。而如果采用 MATLAB－DSP 集成设计环境(系统级集成环境)，就更加简单了。

1.2.4　MATLAB 功能模拟定点 DSP 运算

由于用 MATLAB 精确模拟 DSP 算法太费时间，在某些情况下，我们可以省去这一步。例如，根据经验，我们对 DSP 实现这种算法已有把握，或者在选择算法时，已经为精度问题留有设计余量。这样，我们可以用 MATLAB 仅仅进行功能模拟，初步验证算法的性能，然后利用 DSP 的开发工具，编写 C、汇编程序，详细测试算法在具体的 DSP 上运行的结果及性能。当然，如果把 MATLAB 功能模拟的结果作为理想值，定点 DSP 的处理结果肯定会有较明显的偏差。

以下的 MATLAB 程序模拟用级数展开求平方根的精度。对比上节精确模拟 DSP 算法的程序，可以看出功能模拟 DSP 算法的 MATLAB 程序要简单得多。

```
N=60;
for i=1:N
  y(i)=0.4+0.6*i/N;
  x=y(i) - 1;
  b5(i)=1+x/2- 0.5*(x/2)^2+0.5*(x/2)^3- 0.625*(x/2)^4+0.875*(x/2)^5; %

b7(i)=1+x/2- 0.5*(x/2)^2+0.5*(x/2)^3- 0.625*(x/2)^4+0.875*(x/2)^5- 16/21*(x/2)^6+33/64*(x/2)^7;
end
ideal=sqrt(y);                %理想结果
figure(1);
subplot(2,1,1);
plot(y,b5,'b',y,b7,'k',y,ideal,'r'); zoom on;grid on;title('兰=5 级，黑=7 级，红=理想');
b5err=(ideal- b5)./ideal;            %相对误差
b7err=(ideal- b7)./ideal;
subplot(2,1,2);
plot(y,b5err,'b',y,b7err,'k');zoom on;grid on; title('相对误差兰=5 级，黑=7 级');
ave=sum(abs(b5err))/N; maxerr=max(b5err);     %统计平均误差和最大误差
```

1.2.5 常用的 MATLAB 工具及函数

MATLAB 基本的工具及函数十分丰富，此外还有扩展的大量软件包，这里只将一些常用于 DSP 设计的函数作简单的介绍。

1．矩阵运算

由于 MATLAB 语言是基于矩阵运算的语言，因此在使用时，应尽可能地使用向量、矩阵进行运算，避免针对矩阵元素的循环操作，以提高软件的运行速度，这样程序的书写也很简洁。

用 MATLAB 语言给矩阵/向量赋值有多种形式：

```
for   i =1: N
    a[i]=i*i;
end
b=cos(2*pi*(1:N)/fs)+j* sin(2*pi*(1:N)/fs);      % b 是一个有 N 个元素的复数向量，j 表示复数的
                                                 %虚部
b1(1: N/2)=b(3: 3+N/2- 1);          %从向量 b 的第 3 个元素开始，提取 N/2 个元素形成向量 b1
```

对向量、矩阵的常用运算操作如下：

```
c=a.*b;          %向量 c 等于向量 a 与向量 b 点对点相乘(点乘)
d=a./b;          %向量 c 等于向量 a 与向量 b 点对点相除(点除)
A=［1  2  3  4；5  6  7  8；9  10  11  12］；    %对一个 3×4 矩阵 A 赋值
B=A';            % B 等于 A 的转置矩阵(对复数矩阵则是共扼转置矩阵)
C=A*B;           % C 矩阵等于 A、B 矩阵乘积
```

```
D=inv(C);      % D 矩阵等于 C 矩阵的逆
E=C/D          % 相当于 E=C*D^-1
F=C\D          % 相当于 F=C^-1*D
```

2. 辅助命令□

(1) help 命令：MATLAB 提供了大量的函数和命令，为了便于掌握，MATLAB 提供了联机帮助功能，利用 help 命令，用户可以很容易地得到所需函数的帮助信息。

(2) whos 命令：列出已使用的各种变量、数组；

(3) lookfor 关键词查询命令：在 MATLAB 命令窗口键入“lookfor'关键词'”命令，执行结果是将所有与此关键词有关的命令和函数显示在命令窗口。该命令为初学者提供了极大的便利。

MATLAB 有命令记忆功能，在 MATLAB 命令窗口中，连续使用↑和↓键可以选择在该环境下使用过的命令，不用重新输入这些命令。

3. 普通的滤波器设计方法□

MATLAB 语言包括丰富的滤波器类函数，其中有 FIR 和 IIR 滤波器的各种设计方法以及用设计的滤波器对输入数据进行相应的滤波处理等。比较简单又常用的主要函数有：

(1) fir1 函数：该函数采用不同类型的窗函数(包括矩形窗、Hanning 窗、Hamming 窗、Blackman 窗等)设计具有分段常数幅度特性的 FIR 滤波器，这样 FIR 滤波器具有严格的线性相位。例如：

B = fir1(N, Wn, blackman(N+1))□

在该命令中，N 为设计的滤波器阶数，Wn 为滤波器的归一化截止频率，且 0<Wn<1.0。该函数利用 Blackman 窗设计 N 阶具有线性相位的低通 FIR 滤波器，滤波器系数存在于向量 B 中。

(2) firls 函数：该函数采用最小二乘误差准则设计线性相位 FIR 滤波器。例如：

B=firls(N, F, A)□

在该命令中，N 为设计的滤波器阶数，F 为期望响应的滤波器的频率点(频率点可以不等间隔)，且 0<F<1.0，A 为期望滤波器在这些频率点上的幅频特性。该函数设计了一个 N 阶、满足期望滤波器特性的线性相位 FIR 滤波器，设计的滤波器系数存在于向量 B 中。

(3) filter 函数：该函数利用设计的数字滤波器对数据进行滤波处理，滤波器系数存放于向量 A、B 中。该函数采用直接 II 型结构，可适用于 IIR、FIR 滤波器。例如：

Y =filter(A, B, X)

在该命令中，X 为待滤波的原始数据，Y 为滤波后的输出数据。

(4) 滤波器变换函数：利用该类函数可以将设计的低通滤波器变换为其它类型(如带通、高通等)的滤波器。例如：

［NUMT, DENT］= lp2hpP(NUM, DEN, Wo)

该命令将原型为 NUM(s)/DEN(s)的低通滤波器变换为截止频率为 Wo、响应为 NUMT(s)/DENT(s)的高通滤波器。

4. 滤波器设计分析工具包 FDAtool

MATLAB 的滤波器设计分析工具包 Filter Design & Analysis Tool (FDATool)是一个功能齐全的工具包，可以设计各种结构、任意参数的 FIR、IIR 滤波器，并且可以分析其幅频响

应、相频响应、冲激相应、阶跃相应、零极点图，还可以将设计出的滤波器系数量化为规定位宽和格式的定点数并进行分析。可以用导出命令“File\export”将设计好的系数存到数据文件和 C 语言头文件中，也可将用其它方法得到的滤波器系数用导入命令“Filter\Import Filter”调入 FDATool 中进行分析。

下面对 FDATool 的主要使用方法进行简单介绍。图 1.1 是 FDATool 工具包的界面。

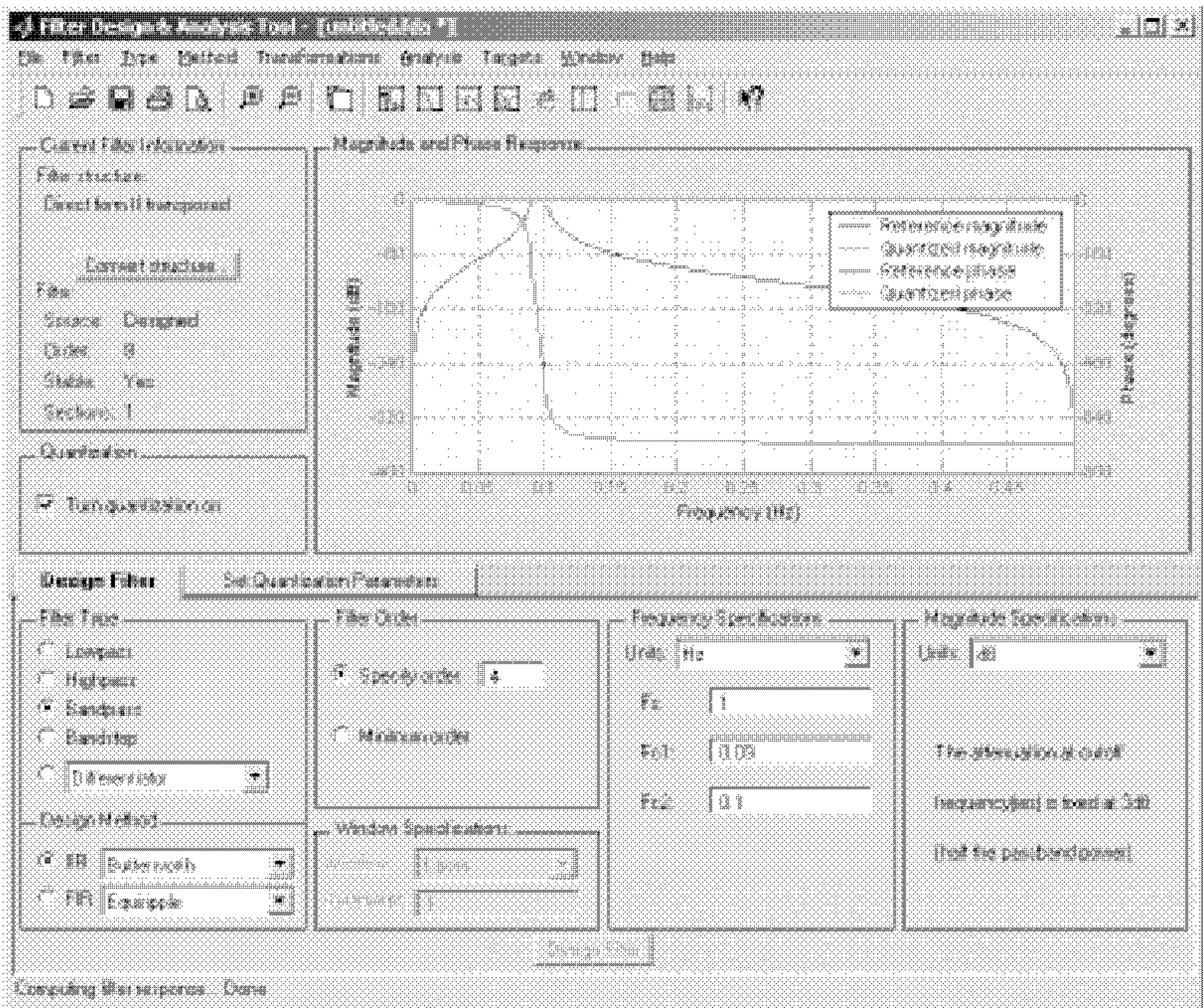


图 1.1　FDATool 工具包的界面

图 1.2 是带通滤波器的设计参数示意图。

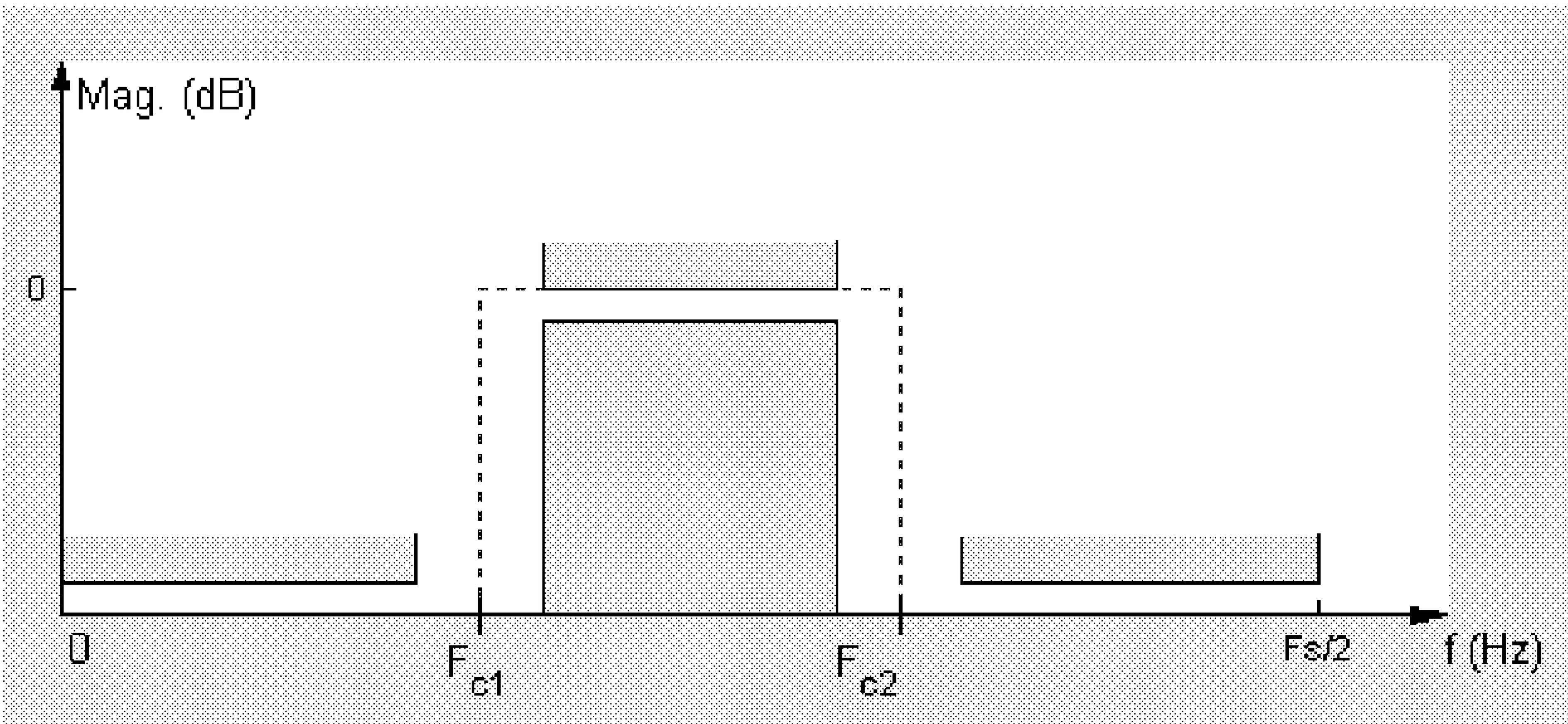


图 1.2　带通滤波器的设计参数示意图

图 1.3 所示的滤波器幅频响应与 1.2.1 节中设计的带通 IIR 滤波器类似，区别在于此滤波器的归一化通带为 0.85～0.125。图中还对滤波器系数进行了 16 位宽的定点格式量化，对应幅频特性较差的曲线。可以看出，量化前后的幅频特性差距很大。图 1.4 是其相频响应。图 1.5 画出了此滤波器的结构图。图 1.6 是其冲激响应。图 1.7 是其零极点图，可见此 IIR 滤波器量化后的极点在单位圆附近，滤波器趋于不稳定。

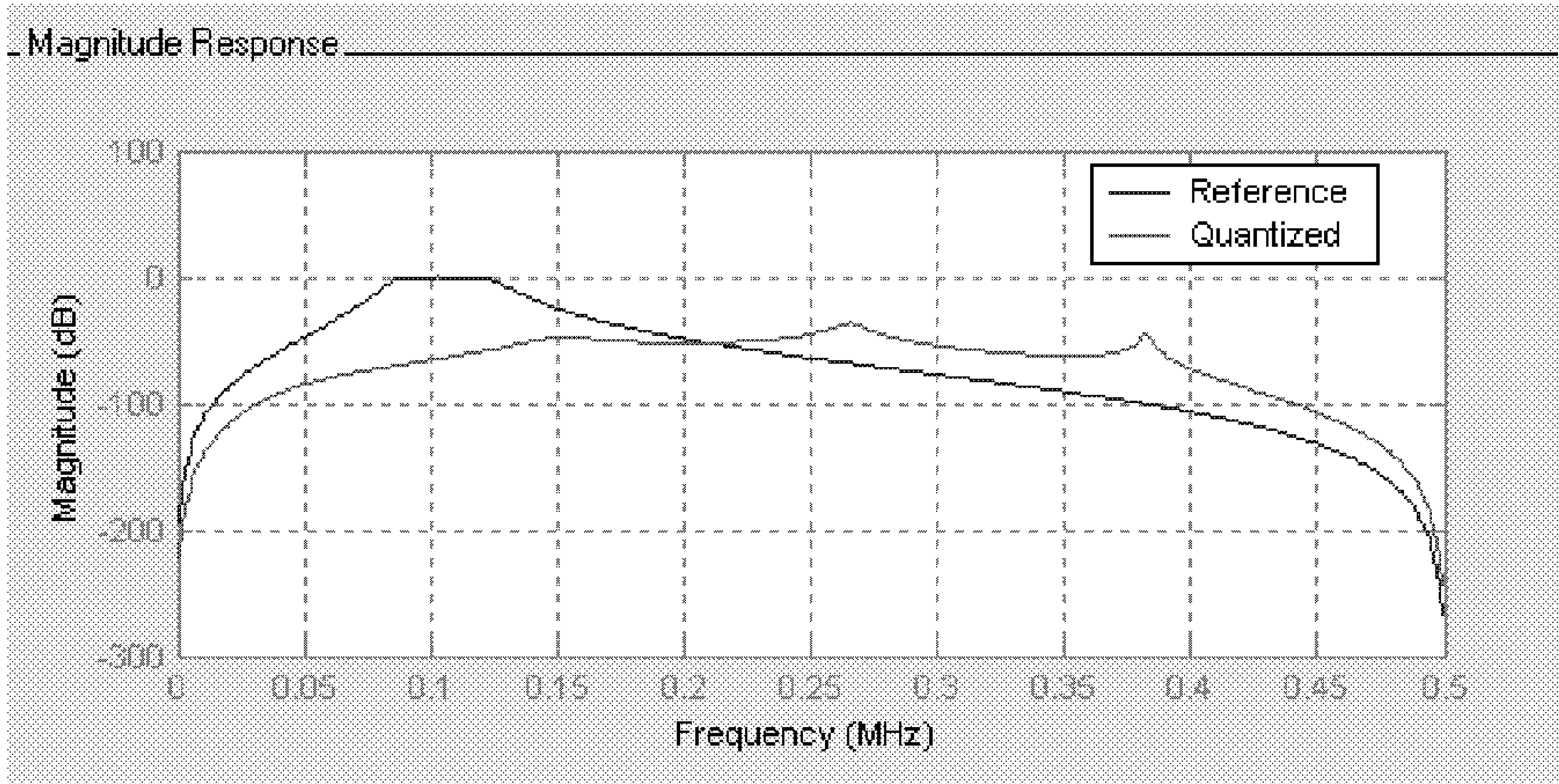


图 1.3　带通 IIR 滤波器的幅频响应

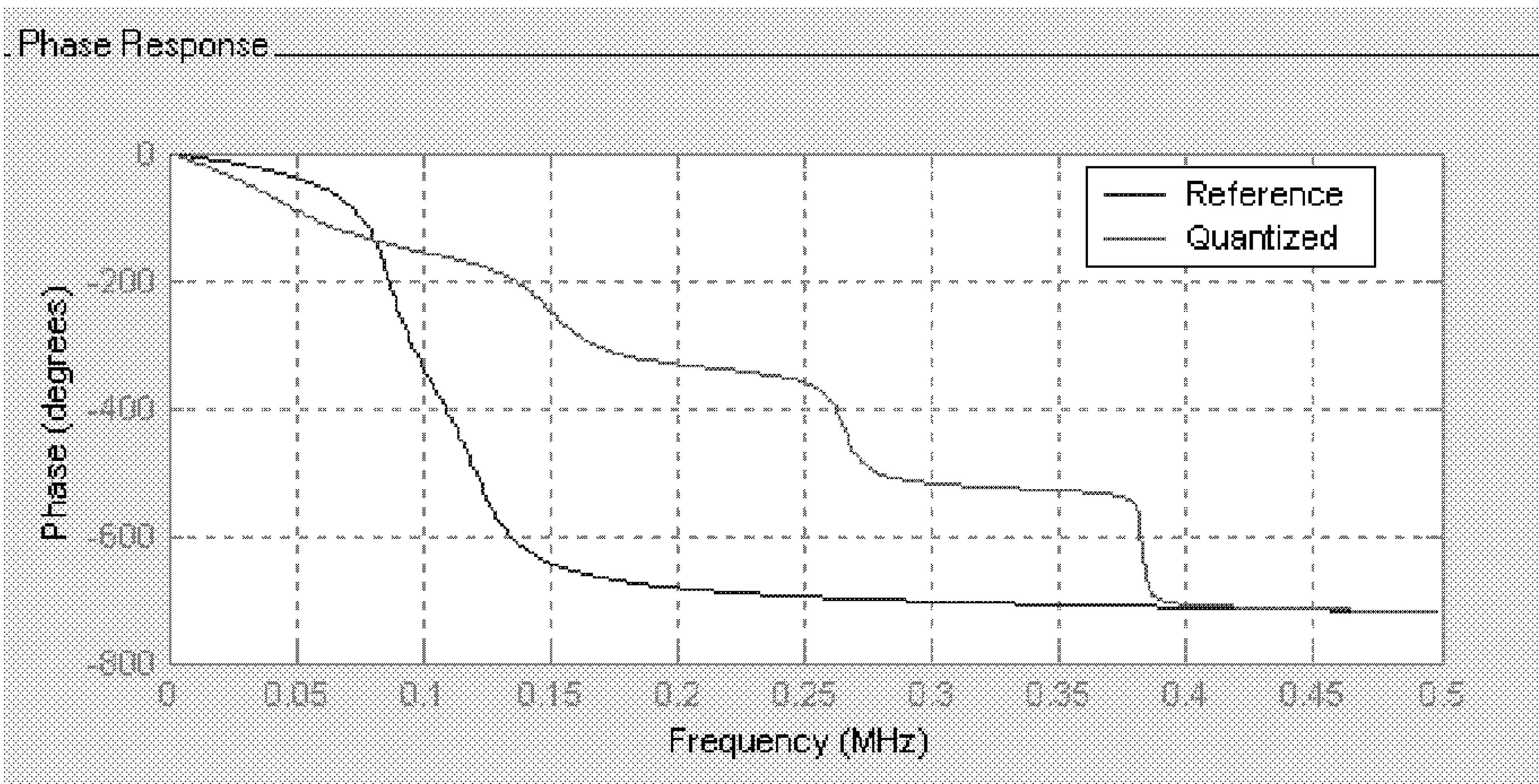


图 1.4　带通 IIR 滤波器的相频响应

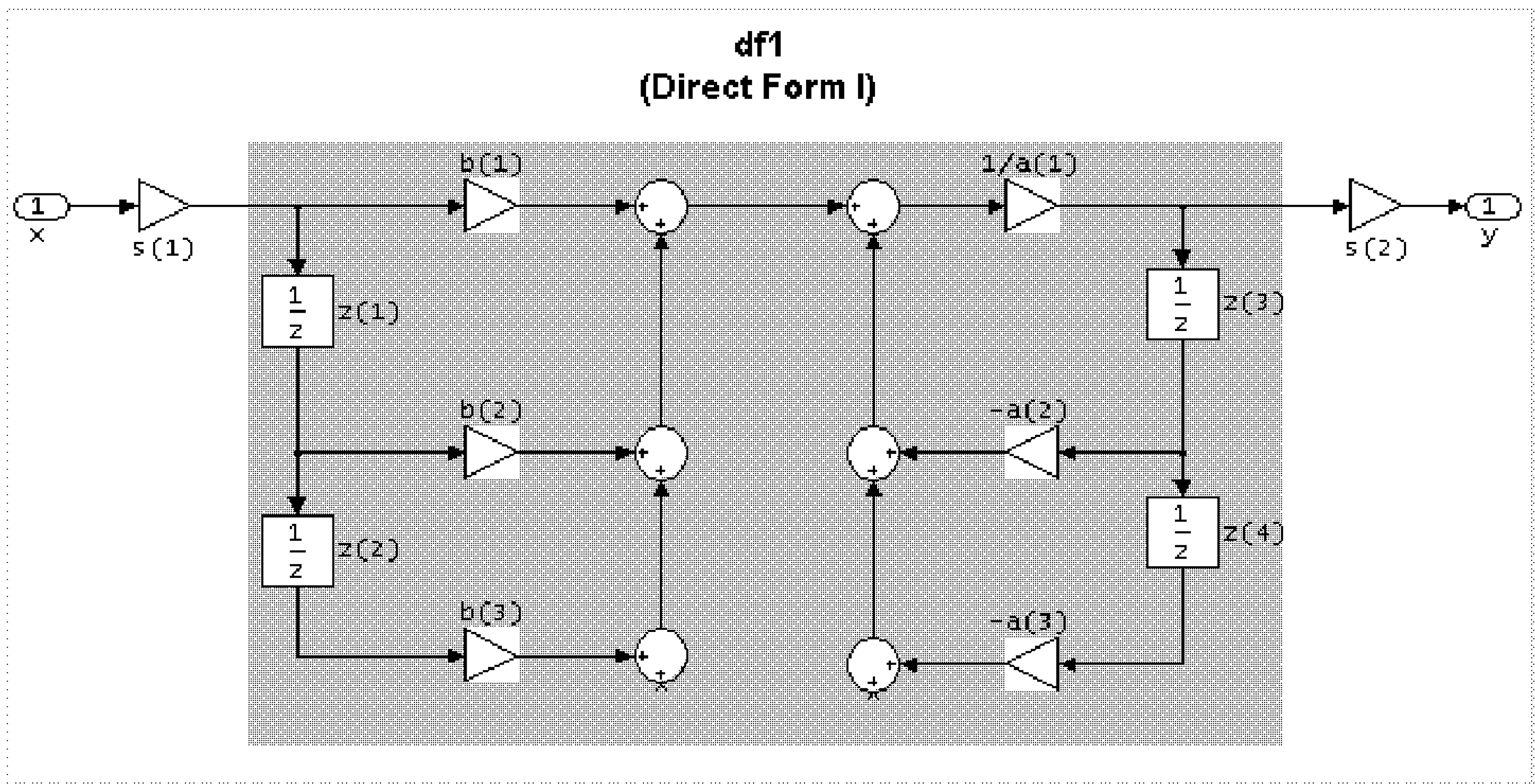


图 1.5　IIR 滤波器的结构图

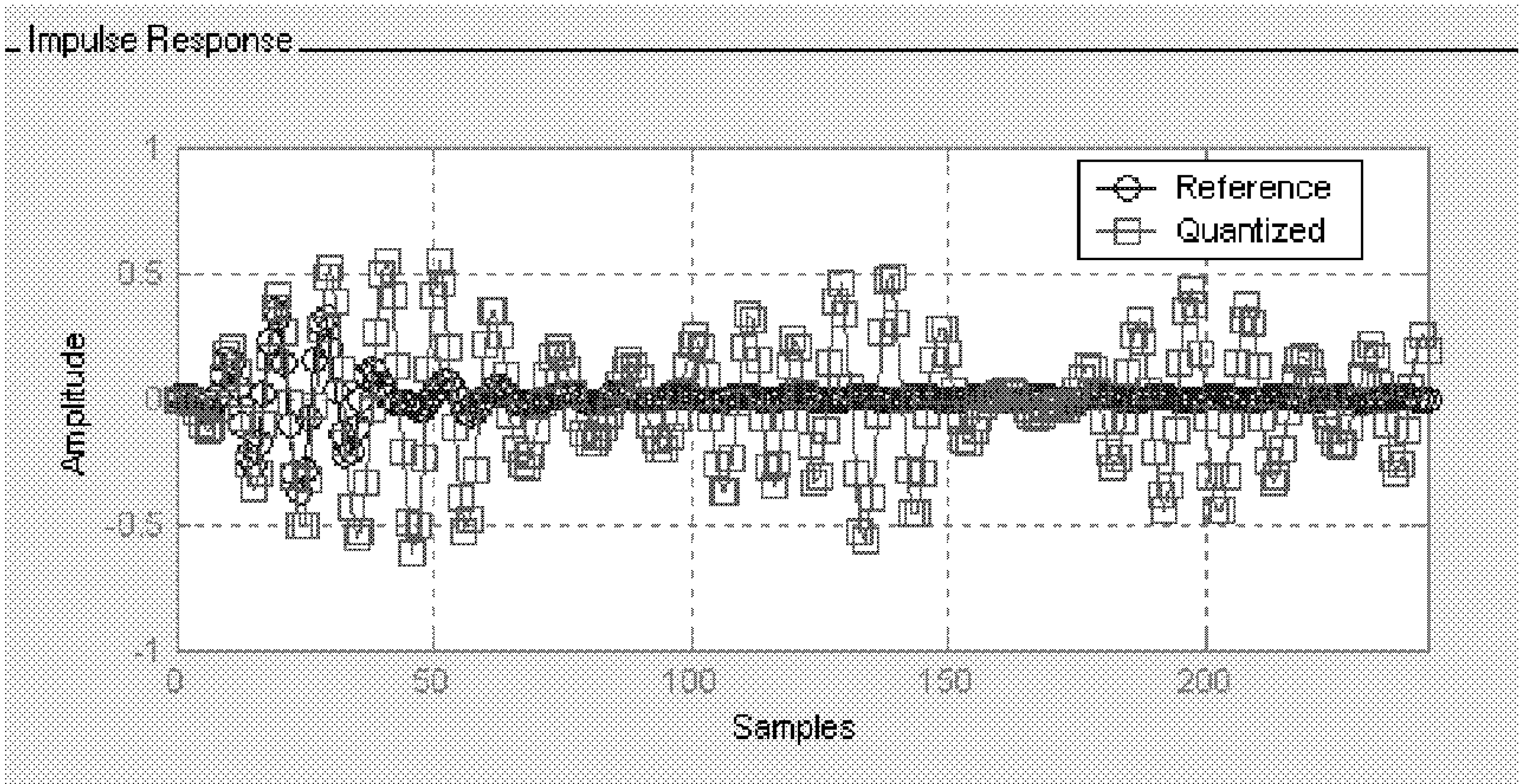


图 1.6　IIR 的冲激响应

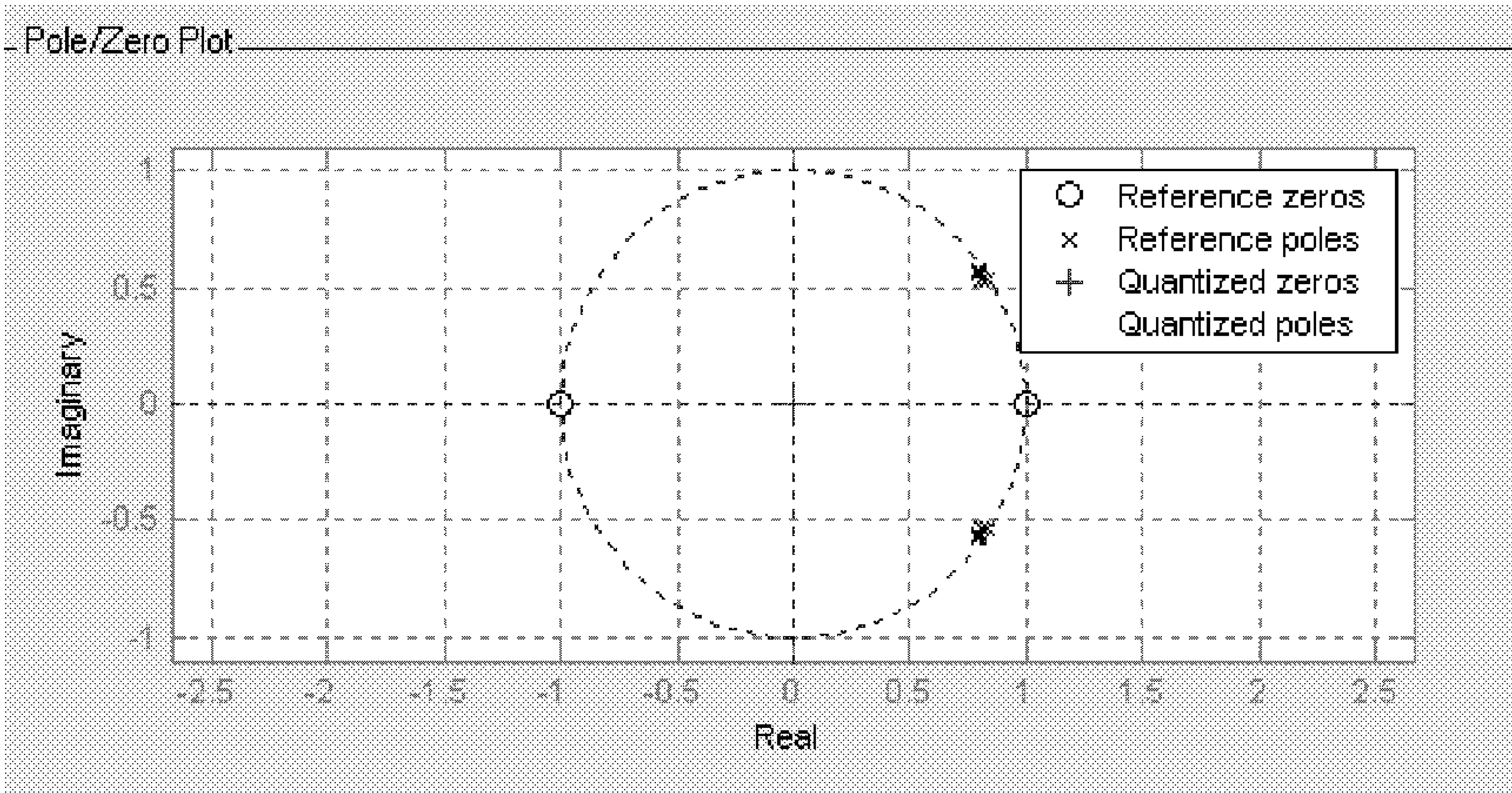


图 1.7　IIR 的零极点图

图 1.8 对滤波器参数进行细微修正，使得原始系数对应的极点远离单位圆，量化后系数的极点也在单位圆内。这是按照 16 位定点格式量化的，若是按照单精度浮点格式截断，效果和稳定性应更好。

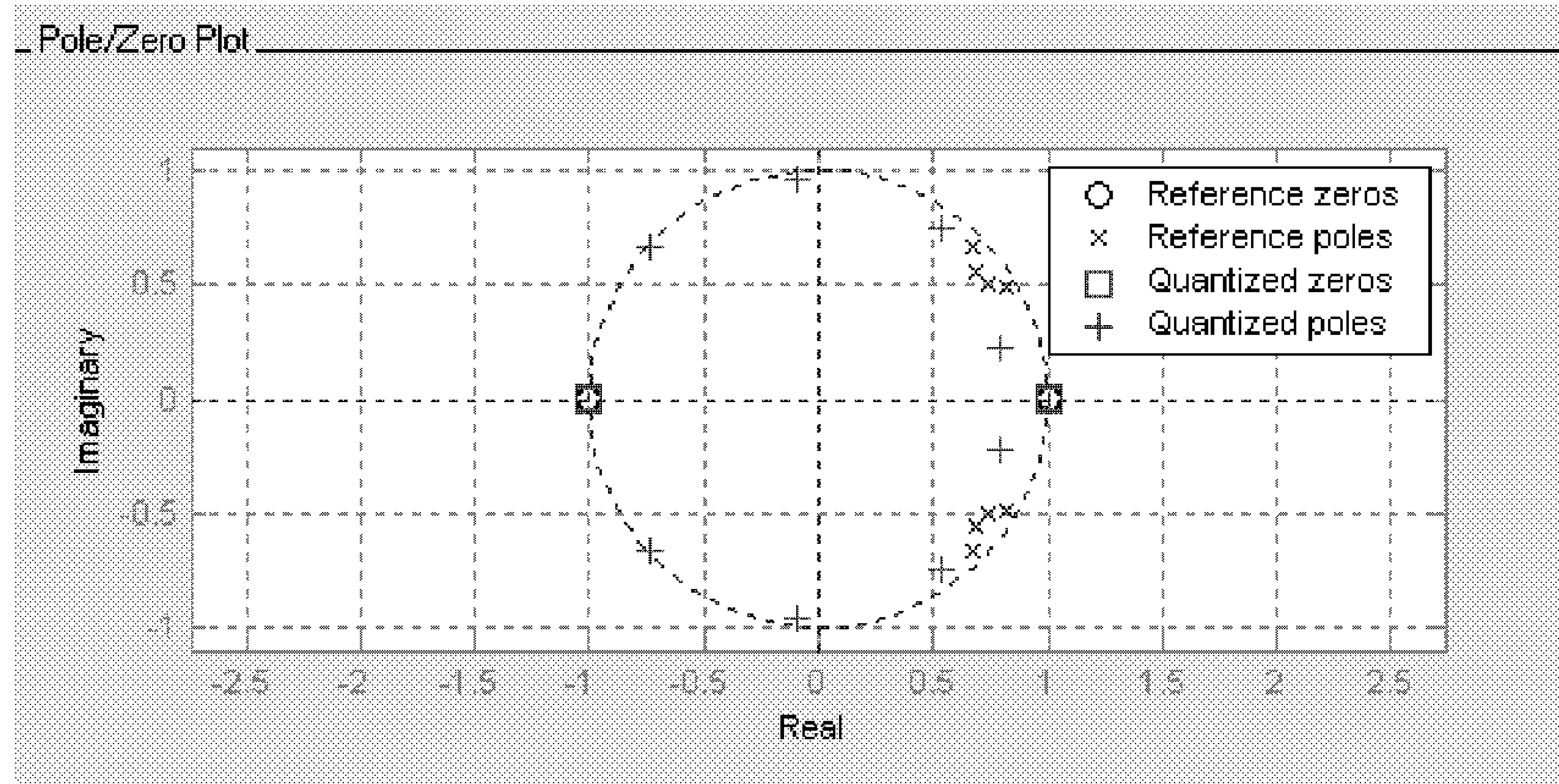
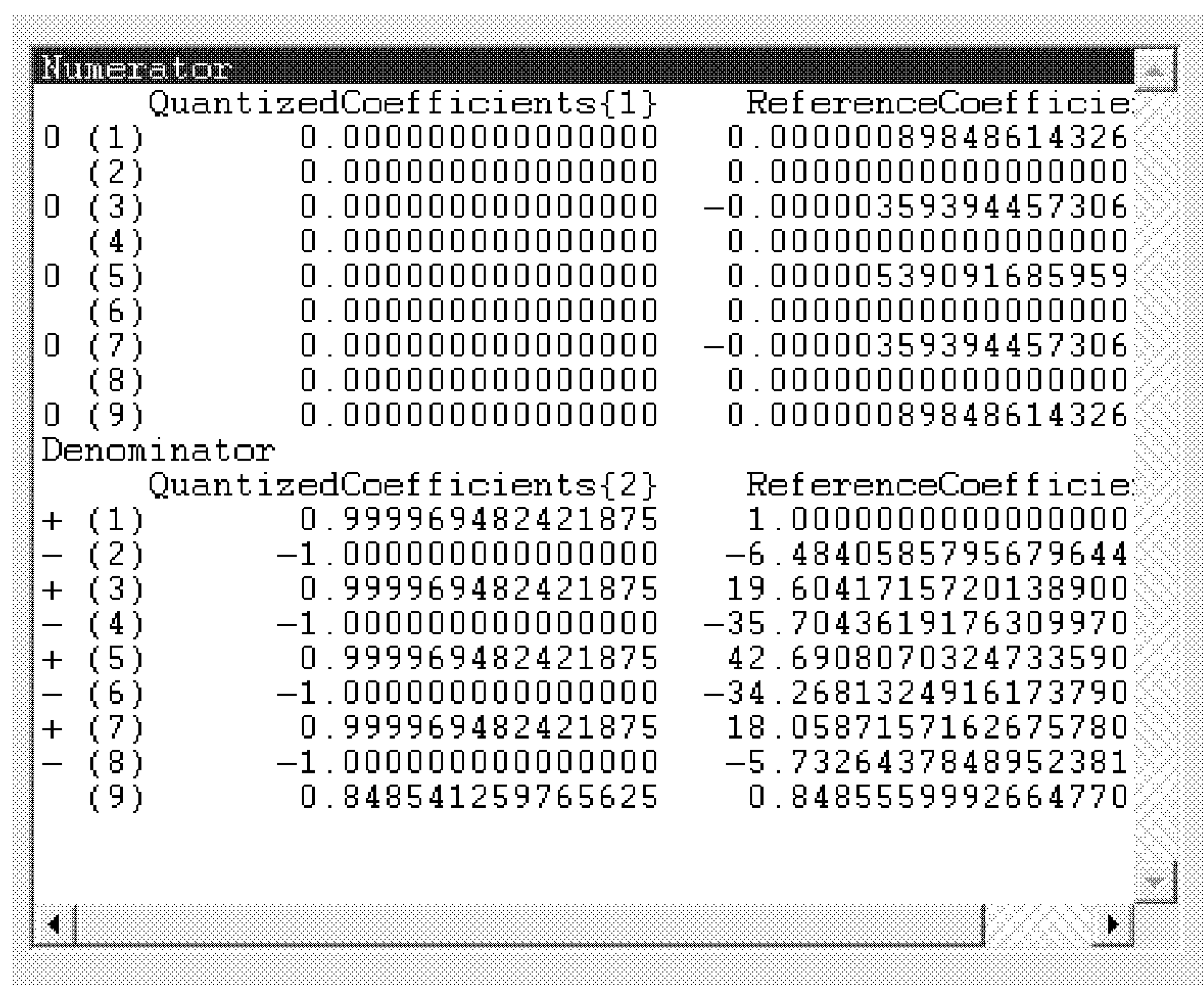


图 1.8　参数细微修正后的 IIR 滤波器的零极点图

如果关心量化前后的滤波器系数，可以将其显示出来，如图 1.9 所示。



```
Numerator
      QuantizedCoefficients{1}    ReferenceCoefficie
0 (1)        0.00000000000000     0.00000089848614326
  (2)        0.00000000000000     0.00000000000000000
0 (3)        0.00000000000000    -0.00000359394457306
  (4)        0.00000000000000     0.00000000000000000
0 (5)        0.00000000000000     0.00000539091685959
  (6)        0.00000000000000     0.00000000000000000
0 (7)        0.00000000000000    -0.00000359394457306
  (8)        0.00000000000000     0.00000000000000000
0 (9)        0.00000000000000     0.00000089848614326
Denominator
      QuantizedCoefficients{2}    ReferenceCoefficie
+ (1)        0.99996948242l875     1.00000000000000000
- (2)       -1.00000000000000    -6.4840585795679644
+ (3)        0.999969482421875    19.6041715720138900
- (4)       -1.00000000000000   -35.7043619176309970
+ (5)        0.999969482421875    42.6908070324733590
- (6)       -1.00000000000000   -34.2681324916173790
+ (7)        0.999969482421875    18.0587157162675780
- (8)       -1.00000000000000    -5.7326437848952381
  (9)        0.848541259765625     0.8485559992664770
```

图 1.9　量化前后的滤波器系数

5. 谱分析

fft 和 ifft 函数：利用这两个函数可以很容易地实现数据的快速付里叶变换和反变换。如：

Y=fft(X, N)

该命令对原始的数据矢量 X 进行 N 点快速付氏变换，N 为任意正整数，省略 N 时，FFT 的点数与 X 的元素个数一致；如果 X 的长度小于 N，则此函数将序列 X 的尾部自动补零至长度 N 后再执行 FFT。

6．MATLAB 的可视化类函数

MATLAB 具有很强的图形绘制能力，主要的函数有：

(1) plot 函数：该函数以线性方式，用不同线型(包括实线、虚线、点划线等)、不同颜色绘制线图。相关的命令如下：

plot(X, Y)：以 X 为横坐标绘制出 Y 的曲线。

grid on：显示网格。

zoom on：允许放大查看图形的局部。

Hold on：在同一图中再画曲线，原来曲线保持不变。

plot(20*log10(abs(Y)))：以对数形式绘制出复数 Y 的幅值曲线，省略参数 X 后，横坐标默认为从 1 到 Y 的元素个数。

plot(X, Y, 'g--')：该命令以 X 为横坐标用绿色以虚线的方式绘制出 Y 的曲线。

plot(Y1, 'r')；hold on；plot(Y2, 'b')；：在一个窗口中画多个曲线，先用红色绘出 Y1 的曲线，再用蓝色绘出 Y2 的曲线。

(2) 对数坐标绘制函数 loglog(X,Y)：该函数与 plot 函数类似，只不过横、纵坐标均采用对数(以 10 为底)坐标而已。

(3) semilogy(X,Y)：该函数与 plot 函数类似，此时，仅纵坐标是采用对数(以 10 为底)坐标，而横坐标仍采用线性坐标。

(4) subplot 函数：该函数将图形窗口分成多个区域，从而在不同区域绘制子图。看下面的命令：

```
subplot (2, 1, 1), plot(income);
subplot (2, 1, 2), plot(outgo);
```

这一对命令将当前的图形窗口分成上、下两区域，在上区域绘制 income 的曲线，在下区域绘制 outgo 的曲线。

(5) contour 函数：该函数将绘制出数据的等高线图形。如 contour(Z, N)将矩阵 Z 用等高线图的方式绘出，且最大值和最小值之间被分成 N 个等级。

除了这些绘图命令外，MATLAB 还提供了专门的绘图函数，如生成直方图的 hist 函数、建立离散序列数据柄状图的 stem 函数等，以适应不同领域的应用。

7. 用 MATLAB 辅助 DSP 设计

这里用一个例子来说明如何利用 MATLAB 工具加快 DSP 的程序调试。MATLAB 为 DSP 产生 FFT 的输入模拟数据文件 test.dat，以供测试 DSP 程序用。test.dat 必须符合 DSP 所要求的数据格式。DSP 和 MATLAB 都对同一批数据进行处理，然后在 MATLAB 下把 MATLAB 和 DSP 的 FFT 结果进行对比。

可以用两种方法把模拟数据文件 test.dat 加载到 DSP 调试环境中：第一种是 DSP 程序用伪指令 include 包含该数据文件，即 .include test.dat，编译/汇编后程序代码中就包含了这批数据；第二种是在 DSP 调试环境中用存储器加载(Fill)命令，把 test.dat 装入存储器。

加载 test.dat 后运行 DSP 程序，再用存储器存储命令(Dump)把 DSP 的 FFT 结果存储到文件 dsp.dat 中。

MATLAB 程序如下：

```
clear all;              %清除所有变量/数组
close all;              %关闭所有图形窗口
N=128; f=4;             %验证 128 点实数的 FFT
fo=fopen('f: \test.dat', 'w');
for i=1: N              %产生双频率的数据文件，作为验证 DSP 的 FFT 程序的模拟数据
    x(i)=1+cos(2.*pi*i*f/N)+4*cos(2.*pi*4*i*f/N);
    x(i)=x(i)*4096;     %将数据定点化为整数，按照定点 DSP 的数据格式写入文件
    fprintf(fo, '.word  %d\r\n', fix(x(i)));
end
fclose(fo);
XF=fft(x)/N;            %用 MATLAB 对此模拟数据进行 FFT，除 N 是因为定点 DSP 的 FFT 过程
                        %中要将结果缩小 N 倍，以避免溢出
figure(1);              %将 MATLAB 的 FFT 结果之实、虚部画在一幅图上
plot(0: N- 1, real(XF)); hold on; plot(0: N- 1, imag(XF), 'r'); zoom on;
figure(2);              %另开一个图形窗口，画以下内容
load f:\dsp.dat;        %这是 DSP 对数据文件 test.dat 进行 FFT 的结果，数据次序为 I0，Q0，I1，
                        %Q1,…，I 表示实部，Q 表示虚部，dsp 有 256 个元素。下面将其转化
                        %为与 MATLAB 数组 XF 一致的复数格式，形成数组 XF1，以便于比较
```

```
XF1(1: N)=dsp(1: 2: 2*N)+j*dsp(2: 2: 2*N));
err0=(XF1- XF);                    %计算各频率点的误差
err1=abs(XF1- XF)./abs(XF);        %计算各频率点的相对误差
subplot(3, 1, 1); plot(0: N- 1, abs(XF), 'r', 0: N- 1, abs(XF1), 'k'); zoom on ;
%在同一图上将 MATLAB 和 DSP 的结果分别用红、黑曲线绘出
subplot(3, 1, 2); plot(abs(err0)); zoom on    %绘出各频率点的误差
subplot(3, 1, 3); plot(abs(err1)); zoom on    %绘出各频率点的相对误差
```

1.3　MATLAB 下的 DSP 集成设计环境

前面介绍的 MATLAB 辅助 DSP 设计虽然方便了 DSP 的设计，但仍然繁琐、费时。而 MATLAB - DSP 集成开发环境彻底改变了以往的 DSP 设计方法。在此环境下可以完成对目标 DSP 的操作，包括访问 DSP 的存储器和寄存器等，又可利用 MATLAB 的强大工具对 DSP 存储器中的数据进行分析和可视化处理。在此环境下，甚至可以直接把 MATLAB 程序生成目标 DSP 的可执行代码。这使熟悉并依赖于 MATLAB 的 DSP 开发人员能够在 MATLAB 下就能完成 DSP 软件开发的全部过程；而对于熟悉 MATLAB 并开始 DSP 设计的初学者来说，MATLAB - DSP 给他们提供了快速设计 DSP 的捷径；而对于利用 MATLAB 专门研究算法并关心其可实现性的算法研究/分析人员来说，MATLAB - DSP 使他们能立刻验证算法在实际 DSP 系统上的可行性。

有其利必有其弊，完全由 MATLAB 生成的 DSP 代码，效率会较低，代码长度、运行速度可能达不到要求，这就必须通过 DSP 代码优化，才能满足我们的设计需求。目前，MATLAB 自身提供的 DSP 代码优化功能还是很有限的，还得由熟练的 DSP 编程人员，根据 DSP 的资源配置，进行多种方式的尝试才行。尝试的途径包括：MATLAB 语言转换成 C 代码后，对 C 语言的编译优化；代码链接时的资源配置优化；手工编写最耗时的汇编子程序。而且此环境也提供了一些汇编优化的模块，在目标程序中加入这些模块可以提高程序的效率。

本书的第 5、6 和 7 章将详细介绍这种 MATLAB - DSP 集成开发环境。

思　考　题

1.1　利用 DSP 汇编语言、C 语言编写 DSP 处理程序各有何优缺点？

1.2　用 MATLAB　DSP 集成开发环境开发 DSP 软件需要哪些工具包？

1.3　MATLAB 工具是如何辅助 DSP 设计进行算法模拟、产生测试数据和验证 DSP 处理结果的？

1.4　在用定点 DSP 实现一个数字信号处理算法前，用 MATLAB 工具精确模拟 DSP 的处理过程有何难点？

第 2 章　高性能通用 DSP 内部功能结构及源代码开发

本章介绍当前最为流行的几种高性能通用 DSP，包括 TI 公司的 TMS320C5000/C6000 DSP 和 AD 公司的 SHARC DSP，主要针对程序开发人员详细介绍这些 DSP 的片内 CPU 核、寄存器、存储器组织、中断、DMA 数据传送和几个常用的片内外设以及这些 DSP 的指令部分(包括汇编和 C/C++语言)。后面几章再分别详细介绍这些 DSP 的开发工具(CCS 和 VisualDSP++)。

几乎所有通用 DSP 的源代码开发都可以采用两种方法：一种是直接利用其提供的汇编指令编写源代码，然后经汇编器和链接器进行汇编链接后生成目标可执行代码；另一种方法是利用标准 C/C++语言编写源代码，然后经 C/C++编译器、汇编器和链接器进行编译链接，最后生成目标可执行代码。

当前，DSP 芯片的发展速度越来越快，旧型号不断被淘汰，新型号的硬件结构和汇编指令越来越复杂，新的处理算法也变得越来越复杂，而且在一种 DSP 上调试好的汇编程序很难移植到另一种类型的 DSP 上。这一切都使得直接利用汇编语言进行软件开发已不适应当前激烈的 IT 市场竞争。C/C++语言易学易用，与硬件无关(或很小)，而且大部分开发者都已经对 C/C++语言程序开发比较熟悉，因此即使不太熟悉 DSP 的人员也能够利用 C/C++语言快速开发出高效的 DSP 应用程序。C/C++语言代码的缺点是效率不高，代码长。随着 DSP 硬件速度的飞速提高和大容量存储技术的日趋成熟，C/C++语言编程的缺点逐渐因硬件的提高而被弥补。当前流行的编程方法是，利用 C/C++语言编写应用程序的主框架，而对于主要的运算函数用汇编语言编写，而且有些开发商提供了一些优化的汇编函数库，可以在 C/C++程序中直接调用这些汇编函数。本章主要介绍汇编指令及在 C/C++程序中如何调用汇编函数，关于标准 C/C++语言的编程问题可查阅相关参考文献。

当前 DSP 程序开发的另一趋势是把 MATLAB 与 DSP 程序的开发、调试结合起来，甚至可以直接从 MATLAB 的 Simulink 模型生成目标 DSP 的可执行代码，从而大大缩短软件开发的周期，加快产品上市。本书的第 6 章详细介绍了如何从 Simulink 模型生成 TMS320C6000 DSP 的可执行代码的过程；第 5 章详细介绍如何利用 MATLAB 来辅助调试 DSP 代码。关于如何从 Simulink 模型生成 MPC555 的可执行代码，读者可查阅 MATLAB6.5(R13)中的相关文档。第三方开发商 SDL 为 AD 公司的 SHARC DSP 开发了一套工具 DSPdeveloper for SHARC，通过 DSPdeveloper for SHARC 和 MATLAB 相结合，可以直接把 Simulink 模型生成 SHARC DSP 的可执行代码，从而在统一的集成环境下完成设计、仿真、代码产生、调试和运行，这些内容将在第 7 章中介绍。感兴趣的读者也可以登陆到 www.sdltd.com/dspdeveloper 网站，查看更多详细信息。

2.1　TMS320C5000 DSP 的内部功能结构及源代码开发

2.1.1　TMS320C5000 DSP 的功能和结构特点

TMS320C5000 系列 DSP 是目前应用最广泛的定点 DSP，其运算速度快，片内 RAM 较大，种类繁多。其主要功能结构如图 2.1 所示。这一系列新型号的推出速度很快，早期的 TMS320C541/C542 等 5V C54x 系列型号已被目前流行的 TMS320VC5402/VC5410 等低电压 C54xx 系列型号所取代。其指令速度在 100 MIPS 以上，最新的 C55xx 速度更高，片内资源更丰富，有双倍数量的乘法器和加法器。在片内结构上，C54x、C54xx、C55xx 有类似之处，它们共享一套指令集，便于程序源代码的相互移植，但由于片内资源不同，硬件上不兼容，程序移植时还要进行相应修改。大体上说，程序从低档的 C54x 向 C54xx 移植较容易，但要对存储器地址等做调整，这可以在链接命令文件(.cmd)中完成。

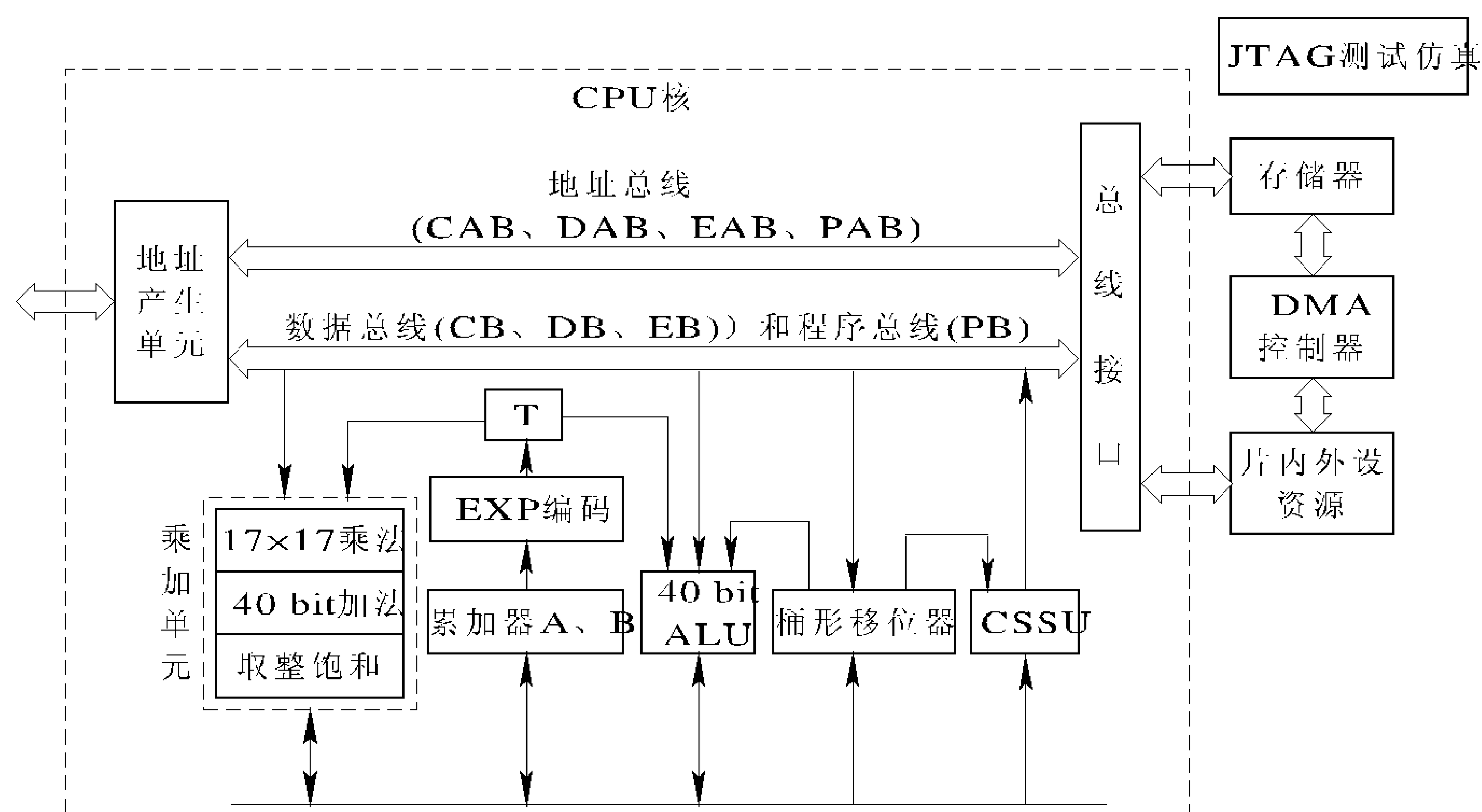


图 2.1　C5000 DSP 内部功能结构

C5000 的特点包括：

• 具有多总线结构。片内有三套数据总线(CB、DB 和 EB)和一套程序总线(PB)以及对应的 4 套地址总线(CAB、DAB、EAB 和 PAB)，4 套总线可同时操作。

• 具有 40 bit 算术逻辑单元 ALU，它包括一个 40 bit 桶形移位器和两个独立的 40 bit 累加器。

• 具有 17 bit×17 bit 乘法器并行一个 40 bit 专用加法器，用于非流水的单周期乘/累加(MAC)运算。

• 具有比较、选择和存储单元(CSSU)，用于 Viterbi 算法。

• 在单周期内可完成 40 bit 累加器值的指数编码(统计多余的符号扩展位)。

• 具有两个地址产生器，包括 8 个辅助寄存器和 2 个辅助寄存器算术单元。

• 某些芯片具有多 CPU 核结构。

• 具有 64 K 字(16 bit)程序(某些芯片可扩展)、64 K 字(16 bit)数据和 64 K 字(16 bit) I/O 的寻址空间；片内存储器配置有很大不同，表 2.1 中列出了不同芯片的片内存储器配置。

- 具有单指令重复或指令块重复功能。
- 具有程序空间和数据空间的数据块移动指令。
- 具有 32 bit 长操作数的指令。
- 一个指令内的 2 或 3 个操作数可以同时读。
- 具有并行存储和并行加载的运算指令。
- 可用 6 级流水完成一条指令操作，6 级流水包括预取指、取指、译码、访问、读数、执行。
- 具有条件存储指令。
- 可从中断快速返回。
- 具有软件堆栈。
- 片内外设资源包括：软等待产生器、可编程的 Bank 切换逻辑、片内锁相环(分频和倍频等功能)、串口、定时器和 DMA 控制器等。表 2.1 中列出了不同芯片的外设资源情况。
- 指令 IDLE1、IDLE2、IDLE3 控制的 3 种低功耗状态，可以禁止 CLKOUT 信号。

表 2.1　C5000 DSP 的片内资源

		C541\LC541\LC541B	C542\LC542	LC543	LC545A\LC546A	LC548	L\VC549	VC5401	U\VC5402	VC5404\VC5407	U\VC5409	VC5410	VC5416	VC5420	VC5421	VC5441	VC5501	VC5502	VC5509	VC5510
片内存储器	RAM(K)	5	10	10	6	32	32	8	16	16/40	32	64	128	200	256	640	16	32	128	160
	ROM(K)	28	2	2	48	2	16	4	4	64/128	16	16	16		4		16	16	32	16
扩展寻址空间						8 M	8 M	1 M	1 M	8 M	8 M	8 M	8 M	256 K	512K		4 M	8 M	8 M	8M
片内外设资源	标准串口	2			1															
	TDM 串口		1	1		1	1													
	缓冲串口		1	1	1	2	2													
	多通道缓冲串口							2	2	3	3	3	3	6	6	12	2	3	3	3
	USB2.0																		√	
	I2C																√	√	√	
	DMA							√	√	√	√	√	√	√	√	√	√	√	√	√
	标准 8 bit 主机口		√		LC545A	√	√													
	扩展 8 bit 主机口							√	√	√	√	√	√				√	√		
	16 bit 主机口									√	√		√	√	√	√		√	√	√
	FIFO													√	√					
	定时器	1	1	1	1	1	1	2	2	2	1	1	1	2	2	8	4	4	2	2
	硬件 PLL	C541\LC54	√	√																
	可编程 PLL	LC541B			√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√

续表

		C541\LC541\LC541B	C542\LC542	LC543	LC545A\LC546A	LC548	L\VC549	VC5401	U\VC5402	VC5404\VC5407	U\VC5409	VC5410	VC5416	VC5420	VC5421	VC5441	VC5501	VC5502	VC5509	VC5510
CPU指令周期/处理速度	25 ns/ 40 MIPS	√	√	√	√															
	20 ns/ 50 MIPS	LC541	LC542	√	√	√		√												
	15 ns/66 MIPS	LC541B			√	√	L													
	12.5 ns/ 80 MIPS						L		U		U									
	10 ns/ 100 MIPS						V		V		V	√								
	10 ns/ 200 MIPS													√	√					
	8.33 ns 120 MIPS									√										
	7.5 ns/ 532 MIPS															√				
	6.94ns 288 MIPS																		√	
	6.25 ns/ 160 MIPS												√							
	6.25 ns/ 320 MIPS																			√
	5.00ns/ 400 MIPS																	√	√	√
	3.33ns/ 600 MPIS																√	√		
	CPU 核数	1	1	1	1	1	1	1	1	1	1	1	1	2	2	4	1	1	1	1

本节接下来以常见的低成本型号 TMS320VC5402 为例介绍 C5000 系列 DSP 的片内结构。由图 2.1 可以看出，TMS320C5000 DSP 的内部结构由三个主要部分组成，即 CPU 核、片内存储器及片内外设资源。

2.1.2 CPU 核

所有 TMS320C54x/C54xx DSP 的 CPU 核都包含如下功能单元：40 bit 算术逻辑单元(ALU)、2 个 40 bit 累加器、桶形移位器、乘/加单元(包含一个 17 bit×17 bit 乘法器和一个 40 bit 加法器)、比较/选择/存储单元(CSSU)指数编码器、寄存器和地址产生单元(包含一个数据地址产生单元和一个程序地址产生单元)。

1．算术逻辑单元(ALU)

ALU 执行基本的二进制补码算术运算和布尔逻辑操作。ALU 的输入可以为：16 bit 立即数、数据存储器中的 16 bit 字、暂存寄存器中的 16 bit 字、数据存储器中的两个 16 bit 字、数据存储器中的 32 bit 字或累加器中的 40 bit 字。

当 ALU 的结果发生溢出时可以设置成饱和模式，由状态寄存器 ST1 的 OVM 位段来设置是否使能饱和模式。状态寄存器 ST0 的 OVA(结果放入累加器 A 中)和 OVB(结果放入累加器 B 中)位段对溢出结果设置标志。

ALU 单元有一个进位位(C)，此位可以被大多数 ALU 的算术指令影响，包括旋转和移位操作。进位位 C 对扩展精度算术操作的高效计算提供支持。两个条件操作码 C 和 NC，用于条件跳转、调用、返回和条件计算。RSBX 和 SSBX 指令可以用来加载进位位。

ALU 单元还可以作为两个 16 bit ALU 使用，能够在单周期内同时执行两个 16 bit 操作。状态寄存器 ST1 的 C16 位段用来设置此模式。

2．累加器 A 和 B

累加器 A 和 B 用来存储 ALU 或乘/加单元的输出结果，它们也可以作为 ALU 单元的第二个(Y)输入数。累加器 A 还可以作为乘/加单元的输入数。每一累加器都可分为如下三部分：保护位(bit39～32)、高位字(bit31～16)和低位字(bit15~0)，分别对应寄存器 AG(BG)、AH(BH)和 AL(BL)。

利用 STH、STL、STLM 和 SACCD 指令或并行存储指令在数据存储器中保存累加器的内容。其中，STH、SACCD 或并行存储指令用来存储累加器的高位字(AH 或 BH)，STL 指令用来存储累加器的低位字(AL 或 BL)。

例　假定累加器 A 的值为 A=FF 4321 1234h，则带移位的累加器存储结果为：

```
STH A, 8,TEMP     ; TEMP = 2112h
STH A, - 8,TEMP   ; TEMP = FF43h
STL A, 8,TEMP     ; TEMP = 3400h
STL A, - 8,TEMP   ; TEMP = 2112h
```

SFTA、SFTL、SFTC、ROL、ROR 和 ROLTC 指令用来对累加器进行移位或旋转。

累加器中的值在存入存储器之前，还可以对其数值采取饱和处理，由 PMST 寄存器的 SST 位段设置是否使能饱和处理。ST1 寄存器的 SXM 设置饱和处理模式，当 SXM=0 时，如果累加器的 40 bit 数值超过 FFFF FFFFh，则会饱和到 FFFF FFFFh；当 SXM=1 时，如果累加器的 40 bit 数值超过 7FFF FFFFh，则会饱和到 7FFF FFFFh，如果累加器的 40 bit 数值小于 8000 0000h，则会饱和到 8000 0000h。最后存储相应的字(高位字、低位字或整个的 32 bit 字)，而累加器中原来的值不会发生变化。

3．桶形移位器

桶形移位器的 40 bit 输入来自累加器、DB 数据总线(16 bit 输入数据)或 DB 数据总线和 CB 数据总线(32 bit 输入数据)。桶形移位器的输出连接到某一 ALU 的输入或 EB 数据总线上。

桶形移位器可用来完成如下定标操作：对输入 ALU 的数据存储器中的数值或累加器中的数值进行定标、对累加器中的数值执行逻辑或算术移位、归一化累加器中的数值、在累加器值保存到数据存储器之前对其进行定标。

桶形移位器和指数编码器可以在单周期内对累加器中的数值进行归一化，输出结果的低位用 0 填充，而高位可以用 0 填充或由符号扩展而来，这由 ST1 寄存器的 SXM 位段进行设置。其它的移位操作还可用来完成数值定标、位抽取和预防溢出等操作。

4．乘/加单元

乘/加单元可以在单周期内执行一个 17 bit×17 bit 二进制补码乘法和一个 40 bit 累加运算。乘/加单元包含如下几个功能块：一个乘法器、一个加法器、无符号/符号输入数控制逻辑、分数控制逻辑、零检测器、取整器、溢出/饱和逻辑和一个 16 bit 暂存器 T。

乘法器具有两个输入数：X 源操作数可以选择暂存器 T、数据存储器(DB 数据总线)或累加器 A(32～16)；Y 源操作数可以选择程序存储器(PB 程序总线)、数据存储器(DB 数据总线或 CB 数据总线)、累加器 A(32～16)或立即数。

加法器的一个输入数来自乘法器的输出结果，另一个来自累加器 A 或 B。

乘/加单元和 ALU 单元可以并行执行，在单周期内完成乘/累加(MAC)和 ALU 操作。

5．比较/选择/存储单元(CSSU)

CSSU 是专用于实现 Viterbi 碟形运算的硬件单元。

CSSU 对累加器的高位字(AH 或 BH)和低位字(AL 或 BL)进行大小比较，并把判决结果 0(如果 AH>AL 或 BH>BL)或 1(如果 AH<AL 或 BH<BL)左移，送入 TRN 寄存器；同时，判决结果也被保存在 ST0 寄存器的 TC 位。根据判决结果，CSSU 把具有最大值的 16 bit 字保存在数据存储器中。利用 CMPS 指令完成上述操作。

例
```
CMPS B,*AR3        ; if (B(31～16)>B(15～0)) then
                   ; B(31～16)->(*AR3); TRN<<1; 0->TRN(0);
                   ; 0->TC}
                   ; else B(15～0)->(*AR3); TRN<<1;
                   ; 1->TRN(0); 1->TC;
```

6．指数编码器

指数编码器是用于专门实现 EXP 指令的硬件单元。指数编码器可以在单周期内执行 EXP 指令。EXP 指令把累加器中数据头的冗余符号位数(减 8 去除保护位)放到暂存器 T 中。NORM 指令再根据 T 中的值对累加器进行移位，以去除冗余的符号位，即可完成对累加器中的值进行归一化操作。

例　对累加器 A 进行归一化：
```
EXP A       ; (the number of leading bits – 8)-> T
NORM A      ; Normalize accumulator A, (A)<<(T)
```

7. 寄存器

TMS320C54x/C54xx 的几乎所有寄存器都映射在片内存储器地址空间中，表 2.2 中列出了常用的 CPU 寄存器和一些片内外设寄存器(以 VC5402 为例，不同 DSP 的外设资源及其寄存器有所不同，用户可以查阅相关资料，本书不再一一介绍)。用户设置这些寄存器时，既可以直接访问相应的存储器地址，也可用寄存器名。所有映射到存储器地址的寄存器统称为 MMR 寄存器(存储器映射寄存器)。

表 2.2　存储器映射寄存器

名　称	地　址(Hex)	说　　明
IMR	0	中断屏蔽寄存器
IFR	1	中断标志寄存器
ST0	6	状态寄存器 0
ST1	7	状态寄存器 1
AL	8	累加器 A 低位字(15～0)
AH	9	累加器 A 高位字(31～16)
AG	A	累加器 A 保护位(39～32)
BL	B	累加器 B 低位字(15～0)
BH	C	累加器 B 高位字(31～16)
BG	D	累加器 B 保护位(39～32)
T	E	暂存器
TRN	F	转换寄存器
AR0～7	10～17	辅助寄存器
SP	18	堆栈指针
BK	19	循环缓冲长度寄存器
BRC	1A	指令块重复计数器
RSA	1B	指令块重复起始地址
REA	1C	指令块重复终止地址
PMST	1D	DSP 模式状态寄存器
XPC	1E	扩展的程序地址指针
DRR20	20	McBSP0 数据接收寄存器 2
DRR10	21	McBSP0 数据接收寄存器 1
DXR20	22	McBSP0 数据发送寄存器 2
DXR10	23	McBSP0 数据发送寄存器 1
TIM	24	定时器 0 减数计数器
PRD	25	定时器 0 周期计数器
TCR	26	定时器 0 控制寄存器
SWWSR	28	软等待状态寄存器
BSCR	29	Bank 间切换控制寄存器
SWCR	2B	软等待控制寄存器
HPIC	2C	主机口控制寄存器

续表

名　　称	地址(Hex)	说　　明
TIM1	30	定时器 1 减数计数器
PRD1	31	定时器 1 周期计数器
TCR1	32	定时器 1 控制寄存器
SPSA0	38	McBSP0 串口 subbank 地址寄存器
SPSD0	39	McBSP0 串口 subbank 数据寄存器
GPIOCR	3C	通用 I/O 控制寄存器
GPIOSR	3D	通用 I/O 状态寄存器
DRR21	40	McBSP1 数据接收寄存器 2
DRR11	41	McBSP1 数据接收寄存器 1
DXR21	42	McBSP1 数据发送寄存器 2
DXR11	43	McBSP1 数据发送寄存器 1
SPSA1	48	McBSP1 串口 subbank 地址寄存器
SPSD1	49	McBSP1 串口 subbank 数据寄存器
DMPREC	54	DMA 通道优先级和使能控制寄存器
DMSA	55	DMA subbank 地址寄存器
DMSDI	56	带增量的 DMA subbank 数据访问寄存器，subbank 地址自动增加
DMSDN	57	不带增量的 DMA subbank 数据访问寄存器
CPKMD	58	时钟模式寄存器

有几个寄存器未映射到存储器地址上，它们是：程序计数器 PC，又称 PC 指针；主机接口寄存器 HPIA 和 HPID，它们只能被主机访问，不能被 DSP 访问。

下面对一些常用的寄存器进行介绍，其它寄存器在后面几节中再专门介绍。

(1) 中断寄存器(IMR 和 IFR)：IMR 用来屏蔽指定的中断，IFR 指示当前中断的挂起情况。IMR 和 IFR 的详细定义在 2.1.4 节中再作介绍。

(2) 状态寄存器(ST0 和 ST1)：包含各种操作状态和模式。表 2.3 和表 2.4 分别列出了 ST0 和 ST1 寄存器的位定义。ST0 寄存器的复位值为 1800h，ST1 寄存器的复位值为 2900h。

表 2.3　状态寄存器 ST0 的位定义

位	名称	说　　明
15～13	ARP	在间接寻址的兼容模式下，选择当前辅助寄存器号 ARx，x=0～7
12	TC	测试/控制标志：受指令 BIT、BITF、BITT、CMPM、CMPR、CMPS 和 SFTC 指令影响；TC 的状态决定条件跳转、调用、运算和返回指令的执行
11	C	进位：当加法的结果产生进位时置 1，当减法的结果产生借位时清零，否则加法会清零此位，减法会置位此位；但是对于带移位的 ADD 和 SUB 指令，ADD 只能置位此位，SUB 只能清零此位，否则不会影响此位；移位和旋转指令以及 MIN、MAX、ABS 和 NEG 指令也会影响此位
10	OVA	累加器 A 溢出标志：当溢出发生时 OVA 置位，直到复位，或带 AOV 和 ANOV 条件的 BC[D]、CC[D]、RC[D]、XC 指令执行时才清零此位，RSBX 指令也可以清除此位
9	OVB	累加器 B 溢出标志：同 OVA
8～0	DP	数据存储器页指针：保存数据存储器空间的高 9 bit 地址，与指令中的低 7 bit 操作数联合形成一个 16 bit 地址，以进行直接寻址；LD 指令可以加载 DP 位段

表 2.4　状态寄存器 ST1 的位定义

位	名 称	说　　明
15	BRAF	块循环激活标志：0=块循环当前无效，1=块循环当前激活，BRC 减为 0 时清除此位
14	CPL	编译方式：0=DP 用于相对直接寻址，1=SP 用于相对直接寻址
13	XF	XF 引脚值：用于控制外部通用输出管脚 XF 的状态，SSBX 指令可以置位 XF，RSBX 指令可以清零 XF
12	HM	保持方式：当 DSP 确认一个有效的 HOLD 信号后，此位指示 DSP 是否继续执行。0=继续执行，但使其外部接口处于高阻状态，1=停止执行
11	TNTM	全局中断屏蔽：0=使能所有可屏蔽中断，1=禁止所有可屏蔽中断，复位值为 1
10	RSVD	保留
9	OVM	溢出方式：选择是否按饱和处理，0=不采取饱和处理，1=采取饱和处理(溢出发生时，保存正最大值 00 7FFF FFFFh 或负最大值 FF 8000 0000h)
8	SXM	符号扩展方式：0=符号扩展禁止，1=符号扩展使能，SSBX 和 RSBX 指令分别用来设置和清除 SXM 位，复位值为 1
7	C16	双 16 bit/双精度算术模式：0=ALU 按双精度模式操作，1=ALU 按双 16 bit 模式操作
6	FRCT	分数模式：1=乘法器输出结果左移 1 位，以补偿额外的符号位
5	CMPT	ARP 工作方式：0=在单存储器操作数的间接寻址模式下，ARP 不被更新，1=在单存储器操作数的间接寻址模式下，ARP 被更新(除了 AR0)
4～0	ASM	累加器移位量：取值范围为−16～15，带并行存储的指令以及 STH、STL、ADD、SUB 和 LD 指令利用此移位值；ASM 可以被 LD 指令加载

(3) 暂存器(T)：可作为乘法和乘/累加指令的一个源操作数，带移位操作指令(例如 ADD、LD 和 SUB 指令)的动态移位量，BITT 指令的动态位地址，DADST 和 DSADT 指令的操作数，还可用来存储 EXP 指令得到的冗余符号位。

(4) 转换寄存器(TRN)：用于保存 CMPS 指令的判决结果(0 或 1)，左移进入 TRN 中。

(5) 辅助寄存器(AR0～AR7)：这 8 个 16 bit 寄存器主要用于为数据空间产生 16 bit 地址，也可作为通用寄存器或计数器。

(6) DSP 模式状态寄存器(PMST)：控制芯片的存储器配置。表 2.5 列出了 PMST 的位定义。

(7) 扩展的程序地址指针(XPC)：包含当前程序地址的高 7 bit。

表 2.5　DSP 模式状态寄存器 PMST 的位定义

位	名 称	复位值	说　　明
15～7	IPTR	1FFh	中断矢量指针，即中断矢量 16 位地址的高 9 位，复位值为 1FFh，相当于指向程序空间的 FF80H 地址处
6	MP/$\overline{MC}$	MP/$\overline{MC}$ 管脚值	DSP 片内 ROM 使能或禁止，0=片内 ROM 使能且可寻址，1=片内 ROM 禁止
5	OVLY	0	片内 RAM 是否映射到程序空间，0=片内 RAM 只映射到数据空间，1=片内 RAM 同时映射到程序和数据空间，但数据页 0(0h～7Fh)不能映射到程序空间
4	AVIS	0	使能/禁止片内程序地址输出到芯片的外部地址管脚上，0=禁止，1=使能

续表

位	名 称	复位值	说　　明
3	DROM	0	使能/禁止片内 ROM 映射到数据空间，0=片内 ROM 不能映射到数据空间，1=部分片内 ROM 可以映射到数据空间
2	CLKOFF*	0	关闭管脚 CLKOUT 输出，1=关闭 CLKOUT 管脚的输出，保持高电平
1	SMUL*	N/A	对乘法器输出结果是否采取饱和处理，1=在 MAC 或 MAS 指令中，乘法器输出结果在进入累加器之前采取饱和处理(OVM 和 FRCT 也必须同时置位；如果 SMUL 没有置位而 OVM 置位，则只对 MAC 或 MAS 指令的最后结果采取饱和处理)
0	SST*	N/A	在存储器保存累加器结果之前是否采取饱和处理，1=存储累加器结果之前采取饱和处理，由 SXM 设置饱和处理模式：当 SXM=0 时，如果累加器的 40 bit 数值超过 FFFF FFFFh，则会饱和到 FFFF FFFFh；当 SXM=1 时，如果累加器的 40 bit 数值超过 7FFF FFFFh，则会饱和到 7FFF FFFFh，如果累加器的 40 bit 数值小于 8000 0000h，则会饱和到 8000 0000h。最后存储相应的字(高位字、低位字或整个的 32 bit 字)，而累加器中原来的值不会发生变化 SST 应用于如下指令：STH、STL、STLM、DST、ST||ADD、ST||LD、ST||MAC[R]、ST||MAS[R]、ST||MPY 和 ST||SUB

注：带*位仅支持 C54x DSP 的修订版 A 或更新的版本，或 C548 以及设备号大于 C548 的 DSP。

2.1.3　存储器组织

TMS320C54x/C54xx 的整个可寻址存储器空间一般为 192 K×16 bit，分成 3 部分：64 K 程序空间、64 K 数据空间和 64 K I/O 空间。而有些 DSP 的程序空间($\overline{PS}$)随地址线的不同而不同，可以扩展到 256 K、1 M 或 8 M。不同 DSP 的片内存储器配置也有很大不同，表 2.1 中列出了不同 DSP 的片内存储器配置以及程序空间的可扩展能力。这种并行结构允许在单周期内完成 4 次存储器访问：一个指令取、两个源操作数读和一个目的操作数写。

TMS320VC5402 有 20 根地址线，其存储器映射空间如图 2.2 所示。PMST 寄存器的三个位 MP/$\overline{MC}$、OVLY 和 DROM 影响存储器的配置。

片内 16 K 字(16 bit)RAM 可同时映射到数据空间和程序空间，由 PMST 的 OVLY 位控制。

片内 ROM 可以被屏蔽掉(MP/$\overline{MC}$=1)，也可以映射到数据存储空间(DROM=1)。片内 ROM 是预先做好的，用户不能改变，但可以使用，其分布如下：

F000～F7FFh　　保留；
F800～FBFFh　　引导程序，上电复位后，DSP 执行此引导程序，将用户代码从外部设备读入，拼装好后放在用户指定的地址；
FC00～FCFFh　　μ 律扩展表；
FD00～FDFFh　　A 律扩展表；

FE00～FEFFh　　sine 表；

FF00～FF7Fh　　保留；

FF80～FFFFh　　中断矢量表，FF80H 是复位中断矢量，DSP 复位后，首先执行 FF80H 的指令。

当 DSP 复位时，若检测到 MP/$\overline{\text{MC}}$ 管脚为低，则程序执行片内 FF80h 地址处的指令，此地址处包含一个跳转到 F800h 的指令，则 CPU 接着执行 F800h 的引导程序，并将区分不同的引导方式，把用户程序代码从外设读来拼装后放在用户指定的地址，然后跳转到用户指定的程序入口处。

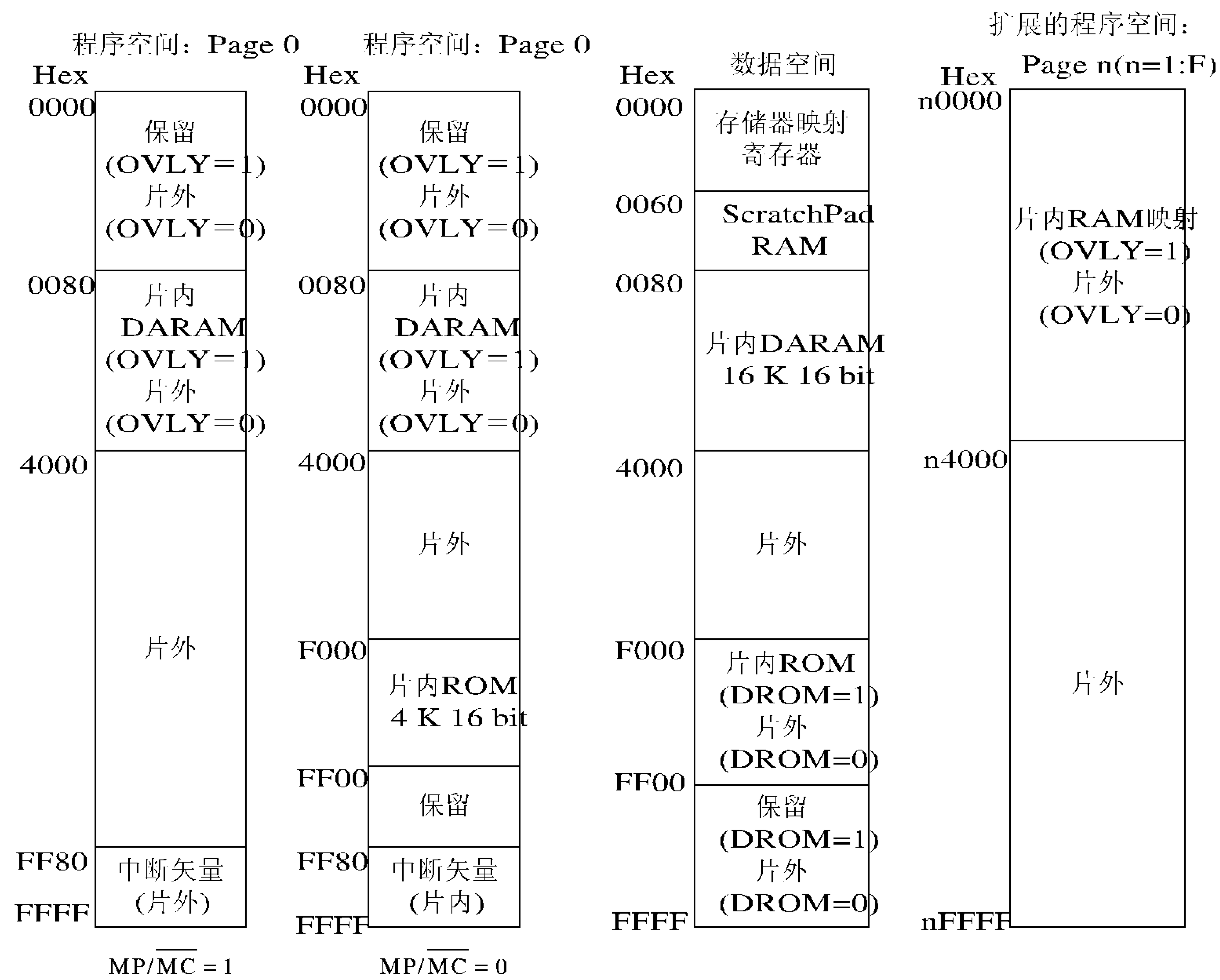


图 2.2　TMS320VC5402 的存储器映射图

2.1.4　中断

TMS320VC5402 支持软件中断和硬件中断，软件中断由指令引起，如 $\overline{\text{INTR}}$、$\overline{\text{TRAP}}$、RESET；硬件中断由外部中断信号或内部中断信号引起，外部硬件中断如 $\overline{\text{INT0}}$～$\overline{\text{INT2}}$，内部硬件中断包括定时器、串口、主机接口等引起的中断。软件中断不分优先级，硬件中断有优先级。硬件中断信号产生后能否引起 CPU 执行相应的中断服务程序，还取决于以下四点(复位、$\overline{\text{NMI}}$ 和软件中断例外，它们不可屏蔽)：

- 状态寄存器 ST1 中全局中断屏蔽位 TNTM 为 0；
- CPU 当前没有响应更高优先级的中断；

- 中断屏蔽寄存器 IMR 中对应的屏蔽位置位；
- 中断标志寄存器 IFR 中对应的中断标志位置位。

VC5402 的中断矢量如表 2.6 所示(SINT 表示软件中断)。

表 2.6　中 断 矢 量 表

中断/陷阱号 K	优先级	名　称	位置(Hex)	功　能
0	1	$\overline{\text{RS}}$ /SINTR	0	软件或硬件复位，优先级最高
1	2	$\overline{\text{NMI}}$ /SINT16	4	不可屏蔽中断
2	—	SINT17	8	软中断 17
3	—	SINT18	C	软中断 18
4	—	SINT19	10	软中断 19
5	—	SINT20	14	软中断 20
6	—	SINT21	18	软中断 21
7	—	SINT22	1C	软中断 22
8	—	SINT23	20	软中断 23
9	—	SINT24	24	软中断 24
10	—	SINT25	28	软中断 25
11	—	SINT26	2C	软中断 26
12	—	SINT27	30	软中断 27
13	—	SINT28	34	软中断 28
14	—	SINT29	38	软中断 29
15	—	SINT30	3C	软中断 30
16	3	$\overline{\text{INT0}}$ /SINT0	40	外中断 0/软中断 0
17	4	$\overline{\text{INT1}}$ /SINT1	44	外中断 1/软中断 1
18	5	$\overline{\text{INT2}}$ /SINT2	48	外中断 2/软中断 2
19	6	TINT0/SINT3	4C	定时器 0 中断/软中断 3
20	7	BRINT0/SINT4	50	串口 0 接收/软中断 4
21	8	BXINT0/SINT5	54	串口 0 接收/软中断 5
22	9	DMAC0/SINT6	58	DMA 通道 0/软中断 6
23	10	TINT1/DMAC1/SINT7	5C	定时器 1 中断/DMA 通道 1/软中断 7
24	11	$\overline{\text{INT3}}$ /SINT8	60	外中断 3/软中断 8
25	12	HPINT/SINT9	64	主机接口/软中断 9
26	13	BRINT1/DMAC2/SINT10	68	串口 1 接收/DMA 通道 2/软中断 10
27	14	BXINT1/DMAC3/SINT11	6C	串口 1 发送/DMA 通道 3/软中断 11
28	15	DMAC4/SINT12	70	DMA 通道 4/软中断 12
29	16	DMAC5/SINT13	74	DMA 通道 5/软中断 13

寄存器 IFR 和 IMR 对各硬中断分别标志和屏蔽。IFR 和 IMR 寄存器的位定义如表 2.7 所示(注意：DMA 通道 0、1、2 和 3 的中断源与其它中断源复用)。

表 2.7　IFR 和 IMR 寄存器的位定义

位	15、14	13	12	11	10	9	8	
功能	保留	DMAC5	DMAC4	BXINT1 DMAC3	BRINT1 DMAC2	HPINT	INT3	
位	7	6	5	4	3	2	1	0
功能	TIN1 DMAC1	DMAC0	BXINT0	BRINT0	TINT0	INT2	INT1	INT0

对 IMR 来说，某位为 1 表示此中断使能，为 0 表示此中断被禁止(屏蔽)。

对 IFR 来说，某位为 1 表示 DSP 收到了一个相应的中断请求(中断挂起)，向 IFR 某位写入 1，清除此中断标志。

中断矢量表可放在程序空间中以 128 字为边界(即低 7 位地址为 0)的任何位置。PMST 寄存器中的中断矢量表指针 IPTR(9 位)形成中断矢量地址的高 9 位，低 7 位由中断矢量号(K 值)乘 4 决定。在 DSP 复位时，IPTR 总为 1FFh，因此复位中断矢量(K＝0)为 FF80H，复位后用户可以改变 IPTR 值来重定位中断矢量表。

当 DSP 响应中断时，将依次完成以下步骤：

(1) 发出 $\overline{\text{IACK}}$ 信号，清除 IFR 中相应的中断标志位(响应可屏蔽中断)和置位 INTM(阻止其它的可屏蔽中断)；

(2) 将 PC 值(返回地址)压入堆栈；

(3) 取中断向量，跳转到相应的中断服务程序；

(4) 保存应保护的寄存器和变量，压入堆栈；

(5) 执行中断处理程序；

(6) 恢复保护的内容，从堆栈弹出；

(7) 中断返回，从堆栈中弹出返回地址；

(8) 继续执行原先的程序。

第(4)、(5)、(6)步要由用户编写程序，在恢复保存的寄存器 BRC、ST1(如果需要保存)时，应注意 BRC 要比 ST1 先恢复。

复位($\overline{\text{RS}}$)是一个不可屏蔽的外中断，复位后 VC5402 的相关内部资源设置成以下状态：IPTR＝1FFh，中断矢量表位于 FF80h，PC＝FF80h；PMST 中的 MP/$\overline{\text{MC}}$ 位与管脚 MP/$\overline{\text{MC}}$ 电平一致；XPC＝0；地址总线置为 FF80h，数据总线为高阻；所有控制线无效；产生 $\overline{\text{IACK}}$ 信号；INTM＝1，屏蔽所有中断；IFR 清 0；置下列状态位的初始值：

ARP＝0	CLKOFF＝0	HM＝0	SXM＝1	ASM＝0	CMPT＝0
INTM＝1	TC＝1	AVIS＝0	CPL＝0	OVA＝0	OVB＝0
BRAF＝0	DP＝0	XF＝1	C＝0	C16＝0	OVM＝0
FRCT＝0	DROM＝0	OVLY＝0			

复位时，由于 SP 没有被初始化，因此用户必须对 SP 进行设置。

2.1.5　片内外设资源

1．直接存储器访问(DMA)

利用 DMA 控制器可以在不影响 CPU 的前提下，实现内部数据存储器、程序存储器和片内外设之间的高速数据传递。VC5402 的 DMA 具有以下特点：

- 有 6 个 DMA 通道，可以实现 6 个独立的数据块传送。
- DMA 的优先级高于 CPU 核。
- 每一个 DMA 通道具有独立的可编程优先级。
- 源/目的地址寄存器在每一数据读/写传送后，可以自动调整(维持不变、增 1、减 1 或用预设值修改)。
- 多帧(Multiframe)传输。传送的每个数据块可以包含多个帧，每个数据块中的帧数和每帧的数据元素数都是可编程的。
- DMA 的每次读或写均可由一个事件触发。
- 每一个 DMA 通道的数据块传送完或传送一半时，可向 CPU 发出一个中断请求。
- 可以实现双字传送(32 bit)。
- 自动初始化(链式 DMA)。每一数据块传送完成后，DMA 通道能够自动初始化并开始进行下一个数据块传送。

1) DMA 寄存器

DMA 通道配置和操作是由一套存储器映射的寄存器完成的。这些寄存器中只有 DMA 通道优先级和使能控制寄存器(DMPREC)是标准存储器映射寄存器，而其它的寄存器都需采用 subbank 寻址模式来访问，即一组寄存器放入一个单存储器地址空间中，通过多路器来访问 subbank 中的某一特定寄存器。表 2.8 中列出了 DMA 的所有寄存器。

表 2.8　DMA 的寄存器

地址(Hex)	子地址(Hex)	名　称	描　　述
54	—	DMPREC	通道优先级和使能控制寄存器
55	—	DMSA	subbank 地址寄存器
56	—	DMSDI	带增量的 subbank 访问寄存器
57	—	DMSDN	不带增量的 subbank 访问寄存器
—	00	DMSRC0	通道 0 源地址寄存器
—	01	DMDST0	通道 0 目的地址寄存器
—	02	DMCTR0	通道 0 数据元素计数器
—	03	DMSFC0	通道 0 同步事件和帧计数寄存器
—	04	DMMCR0	通道 0 传送模式控制寄存器
—	05～09	通道 1 寄存器	通道 1 寄存器，同通道 0
—	0A～0E	通道 2 寄存器	通道 2 寄存器，同通道 0
—	0F～13	通道 3 寄存器	通道 3 寄存器，同通道 0
—	14～18	通道 4 寄存器	通道 4 寄存器，同通道 0
—	19～1D	通道 5 寄存器	通道 5 寄存器，同通道 0

续表

地址(Hex)	子地址(Hex)	名　称	描　　述
—	1E	DMSRCP	源程序页地址(所有通道)
—	1F	DMDSTP	目的程序页地址(所有通道)
—	20	DMIDX0	数据元素地址修改寄存器 0
—	21	DMIDX1	数据元素地址修改寄存器 1
—	22	DMFRI0	帧地址修改寄存器 0
—	23	DMFRI1	帧地址修改寄存器 1
—	24	DMGSA	全局源地址重载寄存器
—	25	DMGDA	全局目的地址重载寄存器
—	26	DMGCR	全局数据元素计数重载寄存器
—	27	DMGFR	全局帧计数重载寄存器

访问 subbank 中某一特定寄存器的方法也很简单：首先把寄存器的子地址写到 subbank 地址寄存器(DMSA)中，并指挥多路器把 subbank 访问寄存器(DMSDI 和 DMSDN)连接到指定的寄存器，然后通过 DMSDI 或 DMSDN 寄存器来访问 subbank 中指定的寄存器。

DMA 控制器提供了两种类型的 subbank 访问寄存器：DMSDI 和 DMSDN。

DMSDI 是子地址会自动增加的 subbank 访问寄存器，即每次访问完一个特定寄存器后，子地址会自动增加。如果需要配置整个 DMA 通道的寄存器，此模式提供了一种方便的方法，无须每次重新指定子地址。如果只需访问某一个特定的寄存器，则利用 DMSDN 寄存器，此 subbank 访问寄存器的子地址不会自动增加。

例　利用 DMSDN(子地址不自动增加)访问 subbank 中的特定寄存器。

下面的代码演示如何把 1000h 值写到 DMA 通道 5 的源地址寄存器(DMSRC5)中。

```
DMSA     .set   55h         ; DMSA 的存储器映射地址
DMSDN    .set   57h         ; DMSDN 的存储器映射地址
DMSRC5   .set   19h         ; DMSRC5 的子地址
STM   DMSRC5, DMSA          ; 把访问存储器的子地址写入到 subbank 地址寄存器(DMSA)中
STM   #1000h, DMSDN         ; 把 1000h 写入到 DMSRC5 中
```

例　利用 DMSDI(子地址会自动增加)访问 subbank 中的寄存器。

下面的代码演示如何配置 DMA 通道 5 的所有寄存器。

```
DMSA      .set   55h        ; DMSA 的存储器映射地址
DMSDI     .set   56h        ; DMSDI 的存储器映射地址
DMSRC5    .set   19h        ; DMSRC5 的子地址
DMDST5    .set   1Ah
DMCTR5    .set   1Bh
DMSFC5    .set   1Ch
DMMCR5    .set   1Dh
STM   DMSRC5, DMSA          ; 把 DMSRC5 的子地址写入到 subbank 地址寄存器(DMSA)中
STM   #1000h, DMSDI         ; 把 1000h 写入到 DMSRC5 中
```

```
STM   #2000h, DMSDI     ; 把 2000h 写入到 DMDST5 中
STM   #0010h, DMSDI     ; 把 10h 写入到 DMCTR5 中
STM   #0002h, DMSDI     ; 把 2h 写入到 DMSFC5 中
STM   #0000h, DMSDI     ; 把 0h 写入到 DMMCR5 中
```

DMA 通道优先级和使能控制寄存器(DMPREC)全局控制 DMA 系统的操作，其作用包括使能每一个 DMA 通道、复路中断选择以及控制通道的优先级。表 2.9 中列出了 DMPREC 寄存器的位定义。

表 2.9　DMA 通道 DMPREC 的位定义

位	名　称	描　述
15	FREE	此位控制在仿真过程中 DMA 控制器的行为：0=当仿真器停止时 DMA 通道的传送暂停，1=当仿真器停止时 DMA 通道的传送继续进行
14	RSVD	保留
13	DPRC[5]	DMA 通道 5 的优先级控制位：1=高优先级，0=低优先级
12～8	DPRC[4]～DPRC[0]	DMA 通道 4～0 的优先级控制位，同 DPRC[5]
7、6	INTOSEL	复路中断选择位： 值　IMR/IFR[6]　IMR/IFR[7]　IMR/IFR[10]　IMR/IFR[11] 00　Reserved　TINT1　BRINT1　BXINT1 01　Reserved　TINT1　DMAC2　DMAC3 10　DMAC0　DMAC1　DMAC2　DMAC3 11　Reserved　Reserved　Reserved　Reserved
5	DE[5]	DMA 通道 5 的使能位：1=使能，0=禁止
4～0	DE[4]～DE[0]	DMA 通道 4～0 的使能位，同 DE[5]

注：DMPREC 的复位值为 0000h。

每一个 DMA 通道可以分别设置为高优先级或低优先级，处于同一优先级别的 DMA 通道按循环优先模式操作。

DMA 通道的源地址寄存器(DMSRCn)和目的地址寄存器(DMDSTn)都是 16 bit 寄存器，它们包含源/目的存储器映射地址的低 16 bit 地址。DMA 控制器在数据传送过程中会自动更新源/目的地址寄存器。

DMA 通道的数据元素计数寄存器(DMCTRn)指定每帧的数据元素数(每帧的最大元素数为 65 536)，在 DMA 通道的每次读传送完成后会自动递减。DMCTRn 的初始化值为每帧中需要传送的数据元素数减 1。在每帧的最后一个数据元素传送完成后，DMCTRn 会自动重载先前的初始值(从其镜像寄存器中得到)。在自动初始化模式下，一旦数据块的最后一帧传送完成后，DMCTRn 会自动被 DMGCR 寄存器重载。 在自缓冲(ABU)模式下，DMCTRn 指定缓冲区的大小。DMCTRn 在传送过程中不会被递减。

DMA 通道同步事件和帧计数寄存器(DMSFCn)具有三个功能：指定触发 DMA 传送的同步事件、指定传送字宽(16 bit 或 32 bit)和指定传送数据块的帧数。表 2.10 中列出了

DMSFCn 寄存器的位定义。

表 2.10　DMA 通道 DMSFCn 的位定义

位	名　称	描　　述
15～12	DSYN[3:0]	指定 DMA 通道的同步事件： 0000=无同步事件，0001=McBSP0 接收事件(REVT0)，0010=McBSP0 发送事件(XEVT0)，0101=McBSP1 接收事件(REVT1)，0110=McBSP1 发送事件(XEVT1)，1101=定时器 0 中断事件，1110=外部中断 3(INT3)事件，1111=定时器 1 中断事件，其它=保留
11	DBLW	选择传送字宽：0=每一数据元素为单字(16 bit)，1=每一数据元素为双字(32 bit)
10～8	rsvd	保留
7～0	FrameCount	指定传送数据块的帧数，初始化值为数据块中的帧数- 1，每数据块的最大帧数为 256，初始化 0 表示单帧；FrameCount 在每帧传送完成后会自动递减 在自动初始化模式下，一旦数据块的最后一帧传送完成，FrameCount 会自动被 DMGFR 寄存器重载

注：DMSFCn 的复位值为 0000h。

DMA 通道的传送模式控制寄存器(DMMCRn)用来控制每一个 DMA 通道的传送模式，其功能包括使能 DMA 自动初始化、选择 DMA 通道中断模式、屏蔽 DMA 中断、源/目的地址的自动修改模式、存储器空间类型和 DMA 传送计数模式。表 2.11 中列出了 DMMCRn 寄存器的位定义。

表 2.11　DMA 通道 DMMCRn 的位定义

位	名　称	描　　述
15	AUTOINIT	DMA 通道的自动初始化模式位：0=自动初始化禁止，1=自动初始化使能
14	DINM	DMA 中断产生屏蔽位：0=屏蔽，1=使能中断产生
13	IMOD	DMA 中断产生模式位： 在 ABU 模式下(CTMOD =1)：0=只有缓冲区满时才产生中断，1=缓冲区一半和满时产生中断； 在多帧模式下(CTMOD =0)：0=数据块传送完成后产生中断，1=在每帧传送完成和整个数据块传送完成时产生中断
12	CTMOD	DMA 传送计数器模式控制位：0=多帧模式，1=ABU 模式
11	rsvd	保留
10～8	SIND	DMA 源地址在传送中自动调整模式位： 000=不调整，001=递增，010=递减，011=按 DMIDX0 寄存器中的预设值调整，100=按 DMIDX1 寄存器中的预设值调整，101=按 DMIDX0 和 DMFRI0 寄存器中的预设值调整，110=按 DMIDX1 和 DMFRI1 寄存器中的预设值调整，111=保留
7、6	DMS	源地址空间类型选择位：00=程序空间，01=数据空间，10=I/O 空间，11=保留
5	rsvd	保留
4～2	DIND	DMA 目的地址在传送中自动调整模式位，同 SIND
1、0	DMD	目的地址空间类型选择位，同 DMS

注：DMMCRn 寄存器的复位值为 0000h。

DMMCRn 寄存器的 SIND(或 DIND)位如果设置为 000，则在整个数据传送过程中地址保持不变；如果设置为 001 或 010，则每个数据元素传送完成后地址自动加 1 或减 1，对于双字(32 bit)字宽传送，地址自动加 2 或减 2；如果设置为 011 或 100，每个数据元素传送完成后，地址由 DMIDX0 或 DMIDX1 中的值进行调整，DMIDX0 或 DMIDX1 的值为 16 bit 符号数，对于双字(32 bit)字宽传送，则地址调整量为 DMIDX0 或 DMIDX1 值的 2 倍；如果设置为 101 或 110，对于帧中的数据元素传送，地址由 DMIDX0 或 DMIDX1 的值进行调整，而对于每帧中最后一个数据元素传送，完成后地址由 DMFRI0 或 DMFRI1 的值进行调整，DMFRI0 或 DMFRI1 的值也是 16 bit 符号整数，利用这种地址调整模式可以方便地实现循环缓冲和数据重排。

一个 DMA 传送数据块包含多个帧，每帧包含多个数据元素。每帧的数据元素数由通道数据元素计数寄存器(DMCTRn)指定，数据块的帧数由 DMSFCn 寄存器的 FrameCount 位段指定。这两个计数器分别在每个数据元素和帧传送完成后自动递减。当一帧的数据元素传送完成后，DMCTRn 会自动重载先前的初始值(从其镜像寄存器中得到)，而 FrameCount 的值减 1；当数据块的最后一个数据元素传送完成后，DMCTRn 和 FrameCount 的值都保持为 0。如果通道的自动初始化模式使能(AUTOINIT=1)，则 DMCTRn 和 FrameCount 分别被 DMGCR 和 DMGFR(低 8 bit)全局重载寄存器重载。DMIDX0 和 DMIDX1 用于对帧中数据元素的传送地址调整，DMFRI0 和 DMFRI1 用于对每帧所有数据元素传送完成后的地址调整。

2) DMA 通道自动初始化(链式 DMA)

DMMCRn寄存器的AUTOINIT位控制DMA通道自动初始化的使能或禁止，AUTOINIT =1 使能自动初始化。DMA 通道的自动初始化只能应用于多帧模式(CTMOD=0)。如果自动初始化使能，则当数据块传送完成后，DMA 通道 n(n=0～5)的下列 4 个寄存器会被自动重载：

源地址寄存器(DMSRCn)被全局源地址重载寄存器(DMGSA)重载；

目的地址寄存器(DMDSTn)被全局目的地址重载寄存器(DMGDA)重载；

数据元素计数器(DMCTRn)被全局数据元素计数重载寄存器(DMGCR)重载；

同步事件和帧计数寄存器(DMSFCn)的 FrameCount 位段被全局帧计数重载寄存器(DMGFR)的低 8 bit 重载，DMGFR 的高 8 bit 保留。

3) DMA 通道同步触发事件

每一个 DMA 通道的数据传送都可以被某一事件来触发，同步事件和帧计数寄存器(DMSFCn)的 DSYN[3:0]位段指定此 DMA 通道的同步事件(如表 2.10 所示)。

4) DMA 通道中断

每一个 DMA 通道都可以向 CPU 发出中断，传送模式控制寄存器(DMMCRn)的 DINM 和 IMOD 位段分别用来设置 DMA 通道的中断使能和选择中断类型(如表 2.11 所示)。然而，DMA 通道 0、1、2 和 3 的中断源与其它中断源复用，因此需要对复路中断进行选择。DMA 通道优先级和使能控制寄存器(DMPREC)的 INTOSEL 位段用来对复路中断进行选择(表 2.9)。

2．定时器

VC5402 有两个定时器，每个定时器带有一个 4 bit 预分频器 PSC 和一个 16 bit 定时计数器 TIM。CLKOUT 时钟先经 PSC 预分频后，用分频的时钟再对 TIM 作减 1 计数，当 TIM

减为 0 时，将在定时器输出脚 TOUT 上产生一个脉冲，同时产生定时器中断请求，并将定时器周期寄存器 PRD 的值装入 TIM。因此，定时器的工作受 3 个寄存器的控制，它们是 TIM、PRD 和定时器控制寄存器 TCR(参见表 2.12)。两个定时器分别具有这 3 个寄存器和相应的输出管脚 TOUT。

表 2.12 定时器控制寄存器 TCR 的位定义

位	名 称	说 明
15～12	rsvd	保留
11	SOFT	SOFT 和 FREE 结合使用
10	FREE	FREE SOFT 定时器操作 0 0 定时器立即停止工作 0 1 计数器减为 0 时停止工作 1 x 定时器继续运行(PRD 重新装入 TIM)
9～6	PSC	预定标计数器，每个 CLKOUT 作减 1 操作，减为 0 时，TDDR 值装入 PSC，TIM 减 1，PSC 的作用相当于预分频器
5	TRB	对此位置 1，将 PRD、TDDR 的值分别装入 TIM 和 PSC，此位读时总为 0
4	TSS	置 0 将启动定时器工作，置 1 将使定时器停止
3～0	TDDR	定时器分频比，以此数对 CLKOUT 分频后再去对 TIM 作减 1 操作，当 PSC 减为 0 时，此值装入 PSC

注：复位后，TCR 的值为 0000h，TIM 和 PRD 均为 FFFFh。

定时器的时间周期按下式计算：

$$CLKOUT\times(TDDR+1)\times(PRD+1)$$

定时器的初始化步骤为：

(1) 将 TCR 中的 TSS 位置 1，关闭定时器；

(2) 修改 PRD；

(3) 重新设置 TCR：令 TRB=1，TSS=0，并按要求设置 SOFT、FREE、TDDR。

利用定时器中断的步骤为：

(1) 将 IFR 中的 TINT 位置 0，清除以前的定时器中断标志；

(2) 将 IMR 中的 TINT 位置 1，打开定时器中断；

(3) 将 ST1 中的 INTM 位置 0，使能所有中断。

每当 TIM 减为 0 时，会产生一个定时器中断，并在相应的 TOUT 管脚上产生一个宽度为 CLKOUT 周期的正脉冲。

3．时钟产生器

时钟产生器为 CPU 提供时钟，有两种参考输入时钟源可供选择：内部振荡器(外接晶振)和外部时钟源。时钟产生器的参考输入时钟再乘以一个比例系数后作为 CPU 的工作时钟。这个比例系数的产生和 VC5402 片内锁相环 PLL 的工作方式有关。VC5402 片内 PLL 是软件可编程的，在 DSP 复位时，它由 3 个管脚 CLKMD1/2/3 的电平决定，这 3 个管脚值也决定了复位时的时钟模式寄存器的值。表 2.13 列出了复位时 CLKMD1/2/3 管脚和时钟模式寄存器 CLKMD 与时钟模式的关系。时钟模式决定了 DSP 工作频率与参考时钟输入的比值。

表 2.13　复位时的时钟模式

CLKMD1	CLKMD2	CLKMD3	CLKMD 寄存器(Hex)	时 钟 模 式
0	0	0	E007	乘 15，内部振荡器工作，PLL 工作
0	0	1	9007	乘 10，内部振荡器工作，PLL 工作
0	1	0	4007	乘 5，内部振荡器工作，PLL 工作
1	0	0	1007	乘 2，内部振荡器工作，PLL 工作
1	1	0	F007	乘 1，内部振荡器工作，PLL 工作
1	1	1	0000	乘 1/2，内部振荡器工作，PLL 不工作
1	0	1	F000	乘 1/4，内部振荡器工作，PLL 不工作
0	1	1	—	保留

VC5402 复位后，通过修改 CLKMD 寄存器就可以重新设置时钟模式。VC5402 有两种时钟模式：

- PLL 模式，其比例系数共 31 种。
- 分频(DIV)模式，其比例系数为 1/2 和 1/4。在此模式下，片内 PLL 电路不工作，以降低功耗。

表 2.14 和表 2.15 表示了如何设置 CLKMD 寄存器以得到希望的比例系数。

表 2.14　时钟模式寄存器 CLKMD

位	名　称	说　　明
15～12	PLLMUL	PLL 乘因子
11	PLLDIV	分频
10～3	PLLCOUNT	PLL 计数值，设定 PLL 启动后需要多少个输入时钟周期，以锁定输出/输入时钟
2	PLLON/OFF	PLL 打开/关闭，此位和 PLLNDIV 都为 0 时，PLL 关闭
1	PLLNDIV	0=工作于分频模式
0	PLLSTATUS	0=表明在 DIV 模式，1=表明在 PLL 模式

注：PLLON/OFF 和 PLLNDIV 共同决定 PLL 是否工作，只有两位都为 0 时，PLL 才不工作。

表 2.15　比例系数与 CLKMD 的关系

PLLNDIV	PLLDIV	PLLMUL	比 例 系 数
0	×	0～14	0.5
0	×	15	0.25
1	0	0～14	PLLMUL+1
1	0	15	1
1	1	偶数	PLLMUL+1+2
1	1	奇数	PLLMUL+4

工作时钟可以通过设置 PMST 寄存器由管脚 CLKOUT 输出，因为 VC5402 的工作频率高达 100 MHz，所以通常关闭 CLKOUT 输出可以减少功耗和噪声。

如果工作过程中需要修改 PLL 比例系数，必须先把时钟产生器转换到 DIV 模式，然后再通过修改 CLKMD 寄存器来改变 PLL 比例系数。同样，也不能直接修改 DIV 模式的比例系数，必须先转换到 PLL 模式才行。相关内容请查阅用户手册，本书不再详细介绍。

4．外部总线访问与等待

VC5402 访问慢速外设时，需要用软等待或硬等待方法来降低对外设的访问速度。硬等待靠外部送到管脚 READY 的信号来实现，软等待则依靠软等待状态寄存器 SWWSR 来设置。SWWSR 将程序空间、数据空间各划分为 2 块，加上 I/O 空间共 5 部分分别设置软等待数。对 TMS320C54xx 系列来说，各 DSP 的速度级别很多，选用零等待外设的标准是外设的访问周期小于 DSP 时钟周期的 60%，否则就应插入等待。

VC4502 对外部总线的访问受 SWWSR、SWCR 和 BSCR 3 个寄存器控制(参见表 2.16 和表 2.17)。

表 2.16　软等待状态寄存器 SWWSR

位	名　称	说　　明
15	XPA	0=程序存储器不扩展，1=程序存储器扩展(大于 64 K)
14～12	I/O	I/O 空间的软等待数(0～7)
11～9	DATA	片外数据空间 8000h～FFFFh 的软等待数(0～7)
8～6	DATA	片外数据空间 0000h～7FFFh 的软等待数(0～7)
5～3	PROG	片外程序空间 X8000h～XFFFFh(XPA=0)的软等待数(0～7)
2～0	PROG	片外程序空间 X0000h～X7FFFh(XPA＝0)或 00000h～FFFFFh(XPA=1)的软等待数(0～7)

复位后，SWWSR=7FFFh，全部片外空间为 7 等待。SWCR 寄存器的最低位(其它位未用)SWSM 为 1 将使 SWWSR 设置的所有等待数乘 2 倍，SWSM 为 0(复位值)乘 1 倍，例如 SWWSR 设置的 7 等待数乘 2 倍后 DSP 实际要按 14 个等待数工作。

表 2.17　组间切换(Bank-Switching)控制寄存器 BSCR

位	名 称	说　　明
15～12	BNKCMP	分组大小。这 4 位分别与地址线 A15～12 对应，决定了分组大小，当两次连续的片外访问在不同组时，会自动插入一个等待 BNKCMP　屏蔽的地址　分组大小 0000　无　64 K 1000　A15　32 K 1100　A15～14　16 K 1110　A15～13　8 K 1111　A15～12　4 K
11	PS－DS	两次连续的片外读访问分别在程序空间和数据空间时，此位为 1 表示要插入一个等待周期，为 0 表示不插入
2	HBH	0=主机总线不保持，1=主机总线保持在先前的电平
1	BH	0=总线不保持，1=总线保持在先前的电平
0	EXIO	0=接通外总线，1=关闭外总线，使地址、数据线为高阻，片选、选通、R/$\overline{W}$、$\overline{MSC}$、$\overline{IAQ}$ 为高电平

注：BSCR 在 DSP 复位后为 F800H。

等待状态意味着 DSP 对片外设备的访问信号线上的有效电平将延长。这些信号线包括地址、R/$\overline{\text{W}}$、$\overline{\text{PS}}$或$\overline{\text{DS}}$或$\overline{\text{IS}}$、$\overline{\text{MSTRB}}$或$\overline{\text{IOSTRB}}$。如果是 DSP 向片外写，数据线的有效电平也要延长。

5．主机接口(HPI)

VC5402 的主机接口包括相应的管脚和 3 个片内寄存器，片外的主机通过修改 HPIC 寄存器来设置工作模式，通过设置寄存器 HPIA 来指定要访问的片内 RAM 单元，通过读写寄存器 HPID 来对指定存储器单元读写。主机通过 HCNTL0、HCNTL1 管脚电平选择 3 个寄存器中的某一个。

VC5402 的主机接口有两种工作方式：

• 共用寻址方式(SAM)：主机和 VC5402 都可以访问片内存储器，异步工作的主机的访问会被 VC5402 的时钟同步，主机与 VC5402 访问冲突时，主机有优先权，VC5402 退让(等待)一个周期。

• 仅主机寻址方式(HOM)：VC5402 处于复位状态或 IDLE2 空闲状态。

主机接口控制寄存器 HPIC(参见表 2.18)既可被主机访问，也可被 VC5402 访问，但 HPIA 和 HPID 只能由主机访问。

表 2.18　主机接口控制寄存器 HPIC

位	名 称	主 机	VC5402	说　明
4	XHPIA	R/W	R	1=主机向 DSP 写的数被存入 HPIA 的高 8 位，0=主机向 DSP 写的数被存入 HPIA 的低 8 位
3	HINT	R/W	R/W	VC5402 向此位写 1，管脚$\overline{\text{HINT}}$=0，向主机发出中断请求
2	DSPINT	W	—	主机向此位写 1，就对 VC5402 提出了主机中断请求，若对其读，读出值总为 0
1	SMOD	R	R/W	1=共用寻址方式，0=仅主机寻址方式，VC5402 复位时 SMOD 为 0
0	BOB	R/W		1=第一个字节为 16 位数(或地址)的低字节，0=第一字节为高字节

对主机来说，HPIC 的高 8 位与低 8 位完全一样，对 VC5402 来说，仅低 8 位有意义。

主机接口的数据线 HD7～0 在关闭主机接口时可用 GPIOCR 寄存器设为通用 I/O 管脚(参见表 2.19)。HD7～0 设置为通用 I/O 管脚后，管脚的电平值(无论作输入还是输出)都反映在通用 I/O 状态寄存器 GPIOSR 的低 8 位的对应位上，可以被读(输入)或写(输出)。GPIOSR 的其它位保留。

表 2.19　通用 I/O 控制 GPIOCR

位	名 称	说　明
15	TOUT1	TOUT1 与 HINT 复用同一管脚，当 TOUT1＝0 时，定时器 1 的输出不外送，当 TOUT1＝1 时，定时器 1 的输出由此管脚输出
7～0	DIR7～0	对应定义 HD7～0 管脚是输出还是输入，当 DIRx＝1 时，HDx 为输出，当 DIRx=0 时，HDx 为输入；若使能主机接口时，DIRx 都被置 0(x=7～0)

主机接口的工作原理简述如下：

(1) 主机利用 HCNTL0、HCNTL1 来区分 3 个 HPI 寄存器，利用 HBIL(字节识别输入线)和 HPIC 寄存器中的 BOB 位区分 16 位数据的高、低字节。因此一种简便的方法是把主机的 3 个低位地址线 A2、A1、A0 分别接到 HCNTL1、HCNTL0、HBIL 上。

(2) 主机先向 HPIC 写入控制字，以设置工作模式，然后将访问地址写入 HPIA，再对 HPID 进行读写，即可读出或写入指定存储单元。当主机连续地把数据写入存储器时，可以只送出一次地址码。VC5402 的主机接口控制逻辑会在每次访问前/后将地址自动增 1。

(3) HRDY 是 VC5402 告诉主机设备已准备好的标志。当 HRDY 为低时，主机应推迟对 VC5402 的访问。但多数情况下，主机访问速度低于 VC5402 反应速度，这时主机可以不理会 HRDY 信号，对 VC5402 连续访问。

(4) 主机和 VC5402 可以分别用 HPIC 中的 DSPINT 和 HINT 位向对方提出中断请求，主机发出的中断请求直接送给 VC5402，而 VC5402 向主机发出的中断请求则送到 $\overline{\text{HINT}}$ 管脚上，只有将 $\overline{\text{HINT}}$ 接到主机的中断源输入上，才能让主机收到这个中断请求。

6．多通道缓冲串口(McBSP)

VC5402 有两个高速多通道缓冲同步串口，其特点包括：全双工，收发双缓冲，带μ 律/A 律压扩，字宽可设为 8/12/16/20/24/32bit，数据时钟和帧信号的速率、极性、内外源均可设置，可设为 TDM(时分复用)的多通道模式，传输率最高为 50 Mb/s。

与串口相关的寄存器很多，可以从 CCS5000 的在线帮助功能获得每个寄存器的详细说明。

当不用串口时，串口 0 和串口 1 可以用管脚控制寄存器 PCR0 和 PCR1 分别设置成通用 I/O 管脚(参见表 2.20)。

表 2.20 管脚控制寄存器 PCRx(x=0, 1)

位	名 称	说 明
15、14		保留
13	XIOEN	0=DX、FSX、CLKX 作串口管脚，1=DX 为通用输出管脚，FSX、CLKX 为通用 I/O 管脚
12	RIOEN	0=DR、FSR、CLKR、CLKS 作串口管脚，1=DR、CLKS 为通用输入管脚，FSR、CLKR 为通用 I/O 管脚
11	FSXM	0=帧同步信号从片外输入，1=由帧同步模式决定
10	FSRM	0=帧同步信号从片外输入，1=由片内产生
9	CLKXM	发送数据时钟模式
8	CLKRM	接收数据时钟模式
7	rsvd	保留
6	CLKS_STAT	CLKS 作为通用输入时的输入电平
5	DX_STAT	DX 作为通用输入时的输入电平
4	DR_STAT	DR 作为通用输入时的输入电平
3	FSXP	FSX 的输入值或输出值
2	FSRP	FSR 的输入值或输出值
1	CLKXP	CLKX 的输入值或输出值
0	CLKRP	CLKR 的输入值或输出值

7．通用 I/O 管脚(GPIO)

VC5402 有两个指定的输入/输出管脚：

XF 是输出管脚，输出值由 ST1 寄存器的 XF 位设置；

BIO 是输入管脚，可用于条件判断指令。

此外，VC5402 的主机接口和串口也可设置为通用 I/O 管脚(参见上文)。这些管脚设置为通用 I/O 后，可以用软件去读取或写值。

8．低功耗方式

VC5402 具有多种低功耗工作方式：指令控制的 IDLE1/2/3 空闲等中断状态、$\overline{\text{HOLD}}$ 控制下的保持状态、关闭外部总线和 CLKOUT 的节能状态。关闭外总线受寄存器 BSCR 控制，关闭 CLKOUT 受寄存器 PMST 控制。

2.1.6　TMS320C5000 DSP 的汇编指令

TMS320C5000 的指令集比较复杂，使用也灵活，用户在使用中可通过集成开发环境 CCS 的在线帮助，得到每条指令的详细解释和示例。C5000 的汇编指令有两种表示法：传统的助记符形式和代数表达式。本书介绍传统的助记符形式。

1．指令形式

指令的基本形式为：

标号：指令　操作数 1，操作数 2，操作数 n

标号是可选项，指示此行代码或变量的存储地址。

指令包括伪指令和处理器指令助记符。伪指令是汇编器指令，用来指挥汇编过程的操作，包括把代码和数据放入指定存储器段、在存储器中保留一段空间用来存放非初始化的变量、声明全局变量和初始化存储器等。表 2.21 中列出了部分伪指令。处理器指令助记符是真正的 DSP 执行命令，它包括算术指令、逻辑指令、程序控制指令和存储指令等。表 2.23 中列出了这些处理器指令助记符。

表 2.21　C5000 编译器的部分伪指令

伪 指 令	说　明
.sect "*name*"	创建数据或代码段
.text	把接下来的代码或数据放入到.text 段(可执行代码段)中
.data	把接下来的数据放入到.data 段(初始化数据段)中
.bss *symbol*, *size in words*	在.bss 段(非初始化数据段)中保留连续 *size* 个字(16 bit)空间，*symbol* 指向保留空间的开始位置
symbol .usect "*section name*", *size in words*	与.bss 的功能类似，但此伪指令重新定义了一个新段 *section name*，*symbol* 指向此段的开始位置
double $value_1$ [, … ,$value_n$] ldouble $value_1$ [, … ,$value_n$]	在当前存储段初始化一个或多个 64 bit IEEE 双精度浮点常数值
.float $value_1$ [, … ,$value_n$] .xfloat $value_1$ [, …,$value_n$]	在当前存储段初始化一个或多个 32 bit IEEE 单精度浮点常数值，但 xfloat 无需把它们指定到 4 字节地址边界上

续表

伪 指 令	说　　明
.long　*value1* [, … ,*valuen*] .xlong　*value1* [, … ,*valuen*]	在当前存储段初始化一个或多个 32 bit 整型常数值，但 xlong 无需把它们指定到 4 字节地址边界上
.ulong　*value1* [, … ,*valuen*]	在当前存储段初始化一个或多个 32 bit 无符号整型常数值
.short　*value1* [, … ,*valuen*] .half　*value1* [, … ,*valuen*] .int　*value1* [, … ,*valuen*] .word　*value1* [, … ,*valuen*]	在当前存储段初始化一个或多个 16 bit 整型常数值，如果伪指令前加 u 字母，例如.ushort，表示初始化 16 bit 无符号整型常数值
.byte　*value1* [, … ,*valuen*] .char　*value1* [, … ,*valuen*]	在当前存储段初始化一个或多个 8 bit 整型常数值，但每一常数值都放入 16 bit 存储单元的低 8 bit。伪指令前加 u 字母，例如.uchar，表示初始化 8 bit 无符号整型常数值
.space *size in bit* .bes *size in bit*	在当前存储段中保留连续的由 *size* 指定的 bit 空间。如果这些伪指令前有标号，则.space 前的标号指向保留空间的第一个字，而.bes 前的标号指向最后一个字
.align *size in word*	把 SPC 指定到 size 字(16 bit)的边界上
.def *symbol1* [, … , *symboln*]	声明当前模块中定义的符号能被其它模块使用
.global *symbol1* [, … ,*symboln*]	声明全局符号
.ref *symbol1* [, … , *symboln*]	声明在当前模块中用到的在其它模块中定义的符号
symbol .set　*value* *symbol* .equ　*value*	定义一个常数符号 *symbol* 来代表 *value*，对于程序中遇到的所有此符号，汇编器自动用 *value* 来代替
.macro　*code*　.endm	定义宏指令

操作数可以没有或有多个，其内容可以是立即数、寄存器、程序地址、数据地址、I/O 地址等。最后一个操作数为目的操作数，如果没有特别指定目的操作数，则最后一个源操作数也作为目的操作数，表 2.23 中列出了每条指令的详细情况。

DSP 进行数据运算时，一般都是通过寄存器进行的，首先把数据装入寄存器，加减运算时用累加器 A 和 B，乘法运算时用累加器和乘数暂存器 T 等，算出的结果再由寄存器传到存储器中。

2．寻址方式

C5000 汇编指令的寻址方式可分为以下 7 种形式：

(1) 立即寻址：操作数是一个立即数，包含在指令中。

例　LD # 12，A　　　；把立即数 12 送到累加器 A，立即数前必须标“#”

(2) 绝对寻址：指令中包含 16 位地址，这种寻址方式为双字指令，速度慢。

例　STL　B，*(y)　　　；把累加器 B 的低 16 位存到变量 y 所在的存储单元中

(3) 累加器寻址：以累加器内容作为地址进行访问，这样地址可以达 20 位(VC5402)或 23 位(LC549 等)，20 位地址将指向程序空间。

例　READA　@x　　　；以累加器 A 的内容作为地址从程序空间取数，放到@x 所指定的数据空
　　　　　　　　　　；间的相应地址上，x 为用户定义的变量

(4) 直接寻址：指令中包含地址的低 7 位，数据地址的高 9 位由 DP 寄存器的低 9 位指定，或者由堆栈指针 SP 指定 16 位地址。这种访问方便快捷，为单字指令，但使用时一定要注意 DP 或 SP 指向当前数据页，每页大小仅 128 字。ST1 的 CPL 位决定选用 DP(CPL=0 时)或 SP(CPL=1 时)。

例　LD　@x，A　　；DP 形成高 9 位地址，@x 形成低 7 位地址，将此 16 位地址所指内容放
　　　　　　　　　；入累加器 A 中；或 16 位地址由 SP 的 16 位值加指令中@x 的 7 位值得
　　　　　　　　　；到，x 为用户定义的变量

编程时容易将直接寻址和立即寻址混淆，从而得不到期望的结果，比较下面两条指令的区别。

```
LD    12h, A
LD    #12h, A
```

第一条指令为直接寻址指令，当页指针由 SP 指定，SP＝0100h 时，表示将数据空间 0100h＋12h＝0112h 单元的内容装入累加器 A 中；而第二条指令为将立即数 12h 装入累加器 A 中，立即数前一定要加“＃”。

(5) 间接寻址：利用辅助寄存器的内容作为地址，同时可以预修改或后修改辅助寄存器值，完成循环寻址和位反序寻址等特殊功能。

例　STL　A，*AR1＋　；把 A 的低 16 位内容放入 AR1 所指单元中，然后 AR1 自动加 1

间接寻址是使用最灵活的寻址方式，表 2.22 中列出了间接寻址的表示方法和功能。

表 2. 22　间接寻址方式(表中 x=0～7)

表示方法	功　能　说　明
*ARx	以 ARx 内容寻址，寻址前后 ARx 内容保持不变
*ARx+	以 ARx 内容寻址，寻址后 ARx 增 1
*ARx-	以 ARx 内容寻址，寻址后 ARx 减 1
*+ARx	ARx 内容先增 1，再寻址
*ARx- 0	以 ARx 内容寻址，寻址后 ARx 内容减去 AR0 的内容
*ARx+0	以 ARx 内容寻址，寻址后 ARx 内容加上 AR0 的内容
*ARx- 0B	以 ARx 内容寻址，寻址后 ARx 内容按位反序减去 AR0 的内容
*ARx+0B	以 ARx 内容寻址，寻址后 ARx 内容按位反序加上 AR0 的内容
*ARx- %	以 ARx 内容寻址，寻址后 ARx 内容按循环寻址方式减 1
*ARx+%	以 ARx 内容寻址，寻址后 ARx 内容按循环寻址方式加 1
*ARx- 0%	以 ARx 内容寻址，寻址后 ARx 内容按循环寻址方式减 AR0 的内容
*ARx+0%	以 ARx 内容寻址，寻址后 ARx 内容按循环寻址方式加上 AR0 的内容
*+ARx(1 k)	先将 16 bit 符号数 1 k 加到 ARx 中，再以 ARx 内容寻址
*ARx(1 k)	以 16 bit 符号数 1 k 和 ARx 内容之和进行寻址，但 ARx 的内容仍保持不变
*+ARx(1 k)%	先将 16 bit 符号数 1 k 按循环寻址方式加到 ARx 中，再以 ARx 内容寻址

当对 32 位(双字)数寻址时，*ARx+、*ARx- 的增/减量是 2 而不是 1。

(6) 存储器映射寄存器寻址。如下两种方法可以用来产生存储器映射寄存器(MMR)的地址：

• 类似于直接寻址，但不管 DP 或 SP 内容为何，高 9 位地址设为 0，低 7 位地址在指令中指定。

• 用间接寻址方式，仍是高 9 位地址置为 0，只用 ARx 的低 7 位，但用于寻址的 ARx 的高 9 位在寻址后被置为 0。

这种方式无论当前 DP 或 SP 指向哪里，都可以直接访问位于地址 0000h～007Fh 的寄存器。这类指令的助记符最后一个字母为 M。

例　LDM MMR，A　　　；将 MMR 寄存器放入 A，MMR 表示任何存储器映射寄存器或地址号，
　　　　　　　　　　　；如 TCR(等效地址为 26h)

对存储器映射寄存器访问的指令有 8 条：LDM、MVDM、MVMD、MVMM、POPM、PSHM、STLM、STM。这些指令的操作数中至少有一个是 MMR 寄存器，MMR 寄存器位于地址为 0000h～005Fh 的数据空间(如图 2.2 所示)，包括了除 PC、HPIA、HPID 以外的所有常用寄存器。

有此指令助记符的最后一个字母虽然也是 M，但它并不是存储器映射寄存器寻址，例如指令“ADDM #1，AR7”，虽然对 MMR 寄存器 AR7 进行了操作，但受 DP 或 SP 影响，属直接寻址操作。当高 9 位地址由 DP 指定时，只有 DP=0 时才是对 AR7 寄存器操作，DP≠0 时则是对指定数据页中偏移量(低 7 位地址)为 17h (AR7 地址为 17h)的存储单元进行直接寻址操作，这一点应特别注意。

(7) 堆栈寻址。当发生中断响应或子程序调用时，PC 会被自动地压入堆栈，堆栈指针 SP 指向存放在堆栈中的最后一个数据，每次压栈前 SP 减 1，每次出栈操作后 SP 加 1。用户可以用 4 条指令来保存和恢复现场数据或传递参数，这 4 种指令为：PSHD，PSHM，POPD 和 POPM。

3．指令表

除了助记符指令形式外，还有一种与之一一对应的代数表达式指令。后者易学易记，易于被 DSP 入门者接受。两种形式的指令不能混用，因为两者的汇编器是不同的。用户可以将早期的汇编程序(只有助记符形式)和新编的代数表达式程序分别作为独立的程序，用两种汇编器分别处理，再将各自生成的 obj 文件链接起来。

表 2.23 中列出了 TMS320C5000 的助记符指令，并简要介绍了这些指令的基本用法和执行操作，表中符号的含义见表后解释。

表 2.23　TMS320C5000 指令集

助记符指令形式		执行操作及描述
算术操作指令		
ADD	Smem, src	(Smem) + (src)→src，单字指令
	Smem, TS,src	(Smem) << (TS) + (src)→src，单字指令
	Smem, 16, src [, dst]	(Smem) << 16 + (src)→(dst 或 src)，单字指令 如果 dst 省略，则 src 也作为目的操作数，以下相同
	Smem[,SHIFT], src[, dst]	(Smem) << SHIFT + (src)→(dst 或 src)，双字指令
	Xmem, SHFT, src	(Xmem) << SHFT + (src)→src，单字指令
	Xmem, Ymem, dst	((Xmem) + (Ymem)) << 16→dst，单字指令

续表(一)

助记符指令形式		执行操作及描述
ADD	# lk [,SHFT], src [, dst]	lk << SHFT + (src)→(dst 或 src)，双字指令
	# lk, 16, src [, dst]	lk << 16 + (src)→(dst 或 src)，双字指令
	src [, SHIFT], [, dst]	(src or [dst]) + (src) << SHIFT→(dst 或 src)，单字指令
	src, ASM [, dst]	(src or [dst]) + (src) << ASM→(dst 或 src)，单字指令
ADDC	Smem, src	(Smem) + (src) + (C) →src，单字指令，符号不扩展
ADDM	# lk, Smem	#lk + (Smem)→Smem，双字指令，此指令不可循环执行
ADDS	Smem, src	uns(Smem) + (src)→src，单字指令，Smem 被当作无符号数
SUB	Smem, src	(src) – (Smem)→src，单字指令
	Smem, TS , src	(src) – (Smem) << TS→src，单字指令，先移位后运算(下同)
	Smem, 16, src [,dst]	(src) – (Smem) << 16→(dst 或 src)，单字指令
	Smem [, SHIFT], src [,dst]	(src) – (Smem) << SHIFT→(dst 或 src)，双字指令
	Xmem, SHFT, src	(src) – (Xmem) << SHFT→src，单字指令
	Xmem, Ymem, dst	(Xmem) << 16 – (Ymem) << 16→dst，单字指令
	# lk [,SHFT], src [,dst]	(src) – lk << SHFT→(dst 或 src)，双字指令
	# lk, 16, src [,dst]	(src) – lk << 16→(dst 或 src)，双字指令
	src [,SHIFT], [,dst]	(src or [dst]) – (src) << SHIFT→(dst 或 src)，单字指令
	src, ASM [, dst]	(src or [dst]) – (src) << ASM→(dst 或 src)，单字指令
SUBB	Smem, src	(src)–(Smem)–(进位 C 的取反值)→src，单字指令，无符号扩展
SUBC	Smem, src	(src) – ((Smem) <<15)→ALU output If　ALU output≥0 　　Then ((ALU output) << 1) + 1→src Else　(src) << 1→src　单字指令
SUBS	Smem, src	(src)–unsigned(Smem)→src，单字指令，Smem 被当作无符号数
MPY	Smem, dst	(T)×(Smem)→dst，单字指令
	Xmem, Ymem, dst	(Xmem)×(Ymem)→dst，(Xmem)→T，单字指令
	Smem, # lk, dst	(Smem)×lk→dst，(Smem)→T，双字指令
	# lk, dst	(T)×lk→dst，双字指令
MPYR	Smem, dst	rnd((T)×(Smem))→dst ，单字指令，rnd 表示对乘积结果取整
MPYA	Smem	(Smem)×(A(32～16))→B，(Smem)→T，单字指令
	dst	(T)×(A(32～16))→dst，单字指令
MPYU	Smem, dst	Unsigned(T)×unsigned(Smem)→dst，单字指令，T 和 Smem 都作为无符号数(相当于两个最高位都为 0 的 17 bit 符号数相乘)
SQUR	Smem, dst	(Smem)→T，(Smem)×(Smem)→dst，单字指令
	A, dst	(A(32～16))×(A(32～16))→dst，单字指令
MAC[R]	Smem,src	(Smem)×(T)+(src)→src，单字指令，带“R”后缀表示对乘加结果取整(下同)
	Xmem, Ymem, src[,dst]	(Xmem)×(Ymem)+(src)→(dst 或 src)，(Xmem)→T，单字指令

续表（二）

助记符指令形式		执行操作及描述
MAC	# lk, src[,dst]	(T)×lk+(src)→(dst 或 src)，双字指令
	Smem,# lk, src[,dst]	(Smem)×lk+(src)→(dst 或 src)，(Smem)→T，双字指令
MACA[R]	Smem [, B]	(Smem)×(A(32～16)) + (B)→B，(Smem)→T，单字指令，B 可以省略但执行操作不变，带“R”表示对乘加结果取整(下同)
	T, src [, dst]	(T)×(A(32～16)) + (src)→(dst 或 src)，单字指令
MACD	Smem, pmad, src	Pmad→PAR If (RC)≠0 Then (Smem)×(由 PAR 寻址的 Pmem)+(src)→src (Smem)→T，(Smem)→Smem +1，(PAR)+1→PAR Else (Smem)×(由 PAR 寻址的 Pmem)+(src)→src (Smem)→T，(Smem)+Smem+1 双字指令，当此指令进入循环时变成单周期执行
MACP	Smem, pmad, src	Pmad→PAR If (RC)≠0 Then (Smem)×(由 PAR 寻址的 Pmem) + (src)→src (Smem)→T，(PAR) + 1→PAR Else (Smem)×(由 PAR 寻址的 Pmem) + (src)→src (Smem)→T 双字指令，当此指令进入循环时变成单周期执行
MACSU	Xmem, Ymem, src	unsigned(Xmem)×signed(Ymem) + (src)→src，(Xmem)→T 单字指令，Xmem 被看作无符号数
MAS[R]	Smem, src	(src) – (Smem)×(T)→src 单字指令，带“R”表示对乘减结果取整
	Xmem, Ymem, src[,dst]	(src) – (Xmem)×(Ymem)→(dst 或 src)，(Xmem)→T，单字指令
MASA	Smem[,B]	(B)–(Smem)×(A(32～16))→B，(Smem)→T，单字指令，B 可以省略但执行操作不变
MASA[R]	T,src[,dst]	(src) – (T)×(A(32～16))→(dst 或 src)，单字指令，带“R”表示对乘减结果取整
SQURA	Smem, src	(Smem)→T，(Smem)×(Smem) + (src)→src，单字指令
SQURS	Smem, src	(Smem)→T，(src)–(Smem)×(Smem)→src，单字指令
DADD	Lmem, src[,dst]	If C16=0 Then (Lmem) + (src)→(dst 或 src) Else (Lmem(31～16)) + (src(31～16))→(dst 或 src) (39～16) (Lmem(15～0)) + (src(15～0))→(dst 或 src) (15～0) 单字指令

续表（三）

助记符指令形式		执行操作及描述
DADST	Lmem, dst	If C16 = 1 Then　(Lmem(31～16)) + (T)→dst(39～16) (Lmem(15～0)) + (T)→dst(15～0) Else (Lmem) + ((T) + (T) << 16)　→dst 单字指令，双字操作数
DRSUB	Lmem, src	If C16 = 0 Then　(Lmem) – (src)→src Else (Lmem(31～16)) – (src(31～16))→src(39～16) (Lmem(15～0)) – (src(15～0))→src(15～0) 单字指令，双字操作数
DSADT	Lmem, dst	If C16 = 1 Then (Lmem(31～16)) – (T)→dst(39～16) (Lmem(15～0)) + (T)→dst(15～0) Else (Lmem) – ((T) + (T << 16))→dst 单字指令，双字操作数
DSUB	Lmem, src	If C16 = 0 Then　(src) – (Lmem)→src Else (src(31～16)) – (Lmem(31～16))→src(39～16) (src(15～0)) – (Lmem(15～0))→src(15～0) 单字指令，双字操作数
DSUBT	Lmem, dst	If C16 = 1 Then (Lmem(31～16)) – (T)→dst(39～16) (Lmem(15～0)) – (T)→dst(15～0) Else　(Lmem) – ((T) + (T << 16))→dst 单字指令，双字操作数
ABDST	Xmem, Ymem	(B) + | (A(32～16)) | →B，((Xmem) – (Ymem)) << 16 → A 单字指令
ABS	src[, dst]	| (src) | →(dst 或 src)，单字指令
DELAY	Smem	(Smem)→Smem+1，单字指令，将数据拷贝到后一个地址
EXP	src	T=src 的符号位个数–8，单字指令

续表（四）

助记符指令形式		执行操作及描述
FIRS	Xmem, Ymem, pmad	pmad→PAR While (RC) ≠ 0 (B) + (A(32～16))×(由 PAR 寻址的 Pmem)→B ((Xmem) + (Ymem)) << 16 →A (PAR) + 1→PAR，(RC) – 1→RC 双字指令，当此指令进入循环时变成单周期执行
LMS	Xmem, Ymem	(A) + (Xmem) << 16 + 2^{15} →A，(B) + (Xmem)×(Ymem)→B 单字指令
MAX	dst	If (A>B) Then (A)→dst，0→C Else (B)→dst，1→C 单字指令
MIN	dst	f (A<B)Then (A)→dst，0→C Else (B)→dst，1→C 单字指令
NEG	src[, dst]	(src)×(–1)→(dst 或 src)，单字指令
NORM	src[, dst]	(src)<<TS→(dst 或 src)，单字指令
POLY	Smem	Round (A(32～16)×(T) + (B))→A，(Smem) << 16→B 单字指令
RND	src[, dst]	(src)+ 2^{15} →(dst 或 src)，单字指令
SAT	src	src 饱和处理，单字指令
SQDST	Xmem, Ymem	(A(32～16))×(A(32～16))+(B)→B，((Xmem) – (Ymem))<<16→A 单字指令
	逻辑操作指令	
AND	Smem, src	(Smem) AND (src)→src，单字指令，按位与，src(39～16)保持不变(下同)
	# lk[, SHFT], src[, dst]	(lk << SHFT)AND (src)→(dst 或 src)，双字指令，先移位再按位与，右移位符号不扩展(下同)
	# lk, 16, src [, dst]	(lk << 16) AND (src)→(dst 或 src)，双字指令
	src [, SHIFT], [, dst]	(dst) AND((src) << SHIFT)→(dst 或 src)，单字指令
ANDM	# lk, Smem	lk AND (Smem)→Smem，双字指令，按位与
OR	与 AND 指令形式类似	按位或
ORM	# lk, Smem	lk OR (Smem)→Smem，双字指令，按位或
XOR	与 AND 指令形式类似	按位异或
XORM	# lk, Smem	lk XOR (Smem)→Smem，双字指令，按位异或
ROL	src	(C)→src(0),(src(30～0))→src(31～1), (src(31))→C, 0→src(39～32) 单字指令，带进位的循环左移 1 位
ROLTC	src	(TC)→src(0),(src(30～0))→src(31～1),(src(31))→C, 0→src(39～32) 单字指令，带 C、TC 的循环左移 1 位
ROR	src	(C)→src(31), (src(31～1))→src(30～0), (src(0))→C, 0→src(39～32) 单字指令，带进位的循环右移 1 位

续表（五）

助记符指令形式		执行操作及描述
SFTA	src, shift [, dst]	(src)<<SHIFT→(dst 或 src)，单字指令，带进位的算术移位
SFTC	src	去掉一个多余的符号位，并设置 TC 位，单字指令
SFTL	src, shift [, dst]	(src)<<SHIFT→(dst 或 src)，单字指令，带进位的逻辑移位
BIT	Xmem, BITC	(Xmem(15 – BITC))→TC，单字指令，复制指定的位到 TC 中
BITF	Smem, # lk	If ((Smem) AND lk) =0 Then 0→TC Else 1→TC，双字指令，测试指定位并由测试结果设置 TC
BITT	Smem	(Smem (15 – T(3～0)))→TC，单字指令，复制指定的位到 TC 中
CMPL	src[, dst]	～(src) →(dst 或 src)，单字指令，按位取反
CMPM	Smem, # lk	If (Smem)=lk Then 1→TC Else 0→TC 双字指令，判断两数是否相等并由结果设置 TC
CMPR	CC，ARx	If (cond) Then 1→TC Else 0→TC，单字指令，比较 ARx 和 AR0 并设置 TC，CC 指定 cond 条件(0=EQ, 1=LT, 2=GT, 3=NEQ)
程序控制指令		
B[D]	pmad	pmad→PC，双字指令，[延迟]跳转，后缀“D”表示延迟跳转，即此指令后的两个单字或一个双字指令也进入流水并执行(下同)
BACC[D]	src	(src(15～0))→PC，单字指令，以 A 或 B 内容作为跳转地址
BANZ[D]	pmad, Sind	若(ARx)≠0，则跳转：pmad→PC，双字指令
BC[D]	pmad，条件 1[，条件 2…]	若条件之一满足则跳转：pmad→PC，双字指令
FB[D]	extpmad	长地址(23 bit)跳转：extpmad(15～0)→PC，extpmad(22～16)→XPC，双字指令
FBACC[D]	src	长地址(23 bit)跳转：(src(15～0))→PC，(src(22～16))→XPC，双字指令
CALA[D]	src	SP 减 1，PC＋1(延迟 PC+3)入栈，src(15～0)→PC，单字指令
CALL[D]	pmad	SP 减 1，PC＋2(延迟 PC+4)入栈，pmad→PC，双字指令
CC[D]	pmad，条件 1[，条件 2，…]	若条件之一满足，则 SP 减 1，PC+2(延迟 PC+4)入栈，pmad→PC，双字指令
FCALA[D]	src	长地址(23 bit)调用：SP 减 1，PC+1(延迟 PC+3)入栈，src(15～0)→PC，src(22～16)→XPC，单字指令
FCALL[D]	extpmad	长地址(23 bit)调用：SP 减 1，PC+2(延迟 PC+4)入栈，extpmad(15～0)→PC，extpmad(22～16)→XPC，双字指令
INTR	K	软件产生中断：SP 减 1，PC+1 入栈，由 K(中断/陷阱号)指定的中断矢量地址→PC，1→INTM，0≤K≤31，单字指令
TRAP	K	进入陷阱：SP 减 1，PC+1 入栈，由 K(中断/陷阱号)指定的中断矢量地址→PC，0≤K≤31，单字指令

续表(六)

助记符指令形式		执行操作及描述
FRET[D]		长地址返回：栈顶内容→XPC，SP 加 1，栈顶内容→XPC，SP 加 1，单字指令
FRETE[D]		长地址返回：栈顶内容→XPC，SP 加 1，栈顶内容→XPC，SP 加 1，0→INTM，单字指令
RC[D]	条件 1[，条件 2，…]	若条件之一满足时返回：栈顶内容→PC，SP 加 1，单字指令
RET[D]		返回：栈顶内容→PC，SP 加 1，单字指令
RETE[D]		返回：栈顶内容→PC，SP 加 1，0→INTM，单字指令
RETF[D]		返回：(RTN)→PC，SP 加 1，0→INTM，单字指令
RPT	Smem	单指令重复，(Smem)→RC，下一条指令的重复次数为(RC)+1(下同)，单字指令
	#K	单指令重复，# K→RC，0≤K≤255，单字指令
	#lk	单指令重复，#lk→RC，0≤lk≤65 535，双字指令
RPTZ	dst, # lk	单指令重复，0→dst，#lk→RC，0≤lk≤65 535，双字指令
RPTB[D]	pmad	指令块重复，块起始地址：PC＋2→RSA(延迟为 PC+4)，块终止地址：pmad→REA(pmad 应为循环块的最后地址–1)，重复次数：(BRC)+1，1→BRAF，双字指令
FRAME	K	(SP) +K→SP，－128≤K≤127，单字指令
POPD	Smem	栈顶内容→Smem，SP 加 1，单字指令
POPM	MMR	栈顶内容→MMR，SP 加 1，单字指令
PSHD	Smem	SP 减 1， Smem→栈顶，单字指令
PSHM	MMR	SP 减 1， MMR→栈顶，单字指令
IDLE	K	空闲状态，等中断，1≤K≤3，单字指令
MAR	Smem	若 CMPT＝0，修改 ARx 若 CMPT＝1 且 ARx≠AR0，修改 ARx，x→ARP 若 CMPT＝1 且 ARx＝AR0，修改 AR(ARP)，单字指令
NOP		空操作，单字指令
RESET		软件复位，单字指令
RSBX	N, SBIT	清除状态寄存器(ST0 或 ST1)的指定位：0→STN(SBIT)，N=0 或 1，0≤SBIT≤15，单字指令
SSBX	N, SBIT	置位状态寄存器(ST0 或 ST1)的指定位：1→STN(SBIT)，N=0 或 1，0≤SBIT≤15，单字指令
XC	n，条件 1[，条件 2，…]	若条件之一满足，则执行后续 n(n=1 或 2)条指令，否则执行 n 个 NOP，单字指令
加载和存储指令		
DLD	Lmem, dst	If C16=0 Then (Lmem)→dst Else (Lmem(31～16))→dst(39～16),(Lmem(15～0))→dst(15～0) 单字指令

续表(七)

助记符指令形式		执行操作及描述
LD	Smem, dst	(Smem)→dst，单字指令
	Smem, TS, dst	(Smem) << TS→dst，单字指令
	Smem, 16 , dst	(Smem) << 16→dst，单字指令
	Smem [, SHIFT], dst	(Smem) << SHIFT→dst，双字指令
	Xmem, SHFT, dst	(Xmem) << SHFT→dst，单字指令
	# K, dst	K→dst，0≤K≤255，单字指令
	# lk [, SHFT], dst	lk << SHFT→dst，双字指令
	# lk, 16, dst	lk << 16→dst，双字指令
	src, ASM [, dst]	(src) << ASM→dst 或 src，单字指令
	src [, SHIFT], dst	(src) << SHIFT→dst，单字指令
	Smem, T	(Smem)→T，单字指令
	Smem, DP	(Smem(8～0))→DP，单字指令
	LD # k9, DP	k9→DP, 0≤k9≤511，单字指令
	LD # k5, ASM	k5→ASM, −16≤k5≤15，单字指令
	LD # k3, ARP	k3→ARP, 0≤k3≤7，单字指令
	LD Smem, ASM	(Smem(4～0))→ASM，单字指令
LDM	MMR, dst	(MMR)→dst(15～0), 00 0000h→dst(39～16), 读存储器映射寄存器，单字指令
LDR	Smem, dst	((Smem) << 16) +(1 << 15)→dst(31～16), Smem 取整放到 dst 的高 16 bit 中，dst 的低 15 bit 清零，bit15 置位。单字指令
LDU	Smem, dst	uns(Smem)→dst(15～0), 00 0000h→dst(39～16),(Smem)作为无符号数，单字指令
LTD	Smem	(Smem)→T，(Smem)→Smem+1，单字指令
DST	src, Lmem	(src(31～0))→Lmem，单字指令
ST	T, Smem	(T)→Smem，单字指令
	TRN, Smem	(TRN)→Smem，单字指令
	# lk, Smem	lk→Smem，双字指令
STH	src, Smem	(src) << (～16)→Smem，单字指令
	src, ASM, Smem	(src) << (ASM～16)→Smem，单字指令
	src, SHFT, Xmem	(src) << (SHFT～16)→Xmem，单字指令
	src [, SHIFT], Smem	(src) << (SHIFT～16)→Smem，双字指令
STL	src, Smem	(src)→Smem，单字指令
	src, ASM, Smem	(src) << ASM→Smem，单字指令
	src, SHFT, Xmem	(src) << SHFT→Xmem，单字指令
	src [, SHIFT], Smem	(src) << SHIFT→Smem，双字指令
STLM	src, MMR	(src(15～0))→MMR，写存储器映射寄存器，单字指令
STM	# lk,MMR	lk→MMR，写存储器映射寄存器，双字指令

续表(八)

助记符指令形式		执行操作及描述
CMPS	src, Smem	If ((src(31～16)) > (src(15～0))) Then (src(31～16))→Smem, (TRN)<<1→TRN, 0→TRN(0), 0→TC Else(src(15～0))→Smem,(TRN)<<1→TRN, 1→TRN(0), 1→TC CSSU 单元执行，用于 Viterbi 算法，单字指令
SACCD	src, Xmem，条件	若条件满足，则(src) << (ASM–16)→Xmem，单字指令
SRCCD	Xmem，条件	若条件满足，则(BRC)→Xmem，单字指令
STRCD	Xmem，条件	若条件满足，则(T)→Xmem，单字指令
ST src, Ymem ‖ LD Xmem, dst		(src) << (ASM –16)→Ymem，(Xmem) << 16→dst 单字指令
ST src, Ymem ‖ LD Xmem, T		(src) << (ASM –16)→Ymem，(Xmem)→T 单字指令
LD Xmem, dst ‖ MAC[R] Ymem,dst_		(Xmem) << 16→dst (31～16),((Ymem)×(T)) + (dst_) _→dst_ 带后缀“R”表示对 MAC 结果取整后再存入 dst_中，单字指令
LD Xmem, dst ‖ MAS[R] Ymem,dst_		(Xmem) << 16→dst (31～16)，(dst_) – ((T)×(Ymem))→dst_ 带后缀“R”表示对 MAS 结果取整后再存入 dst_中，单字指令
ST src, Ymem ‖ ADD Xmem, dst		(src) << (ASM–16)→Ymem，(dst_) +((Xmem) <<16)→dst 注意 dst 和隐含的 dst_，单字指令
ST src, Ymem ‖ SUB Xmem, dst		(src) << (ASM–16)→Ymem，((Xmem) <<16) – (dst_)→dst 注意 dst 和隐含的 dst_，单字指令
ST src, Ymem ‖ MAC[R] Xmem, dst		src << (ASM – 16)→Ymem，(Xmem)×(T) + (dst)→dst 带后缀“R”表示对 MAC 结果取整后再存入 dst 中，单字指令
ST src, Ymem ‖ MAS[R] Xmem, dst		src << (ASM – 16)→Ymem，(dst) – (Xmem)×(T)→dst 带后缀“R”表示对 MAS 结果取整后再存入 dst 中，单字指令
ST src, Ymem ‖ MPY Xmem, dst		src << (ASM – 16)→Ymem，(T)×(Xmem)→dst 单字指令
MVDD	Xmem,Ymem	(Xmem)→Ymem，单字指令
MVDK	Smem,dmad	(dmad)→EAR If (RC)≠0 Then (Smem)→EAR 寻址的数据单元，(EAR)+1→EAR Else (Smem)→EAR 寻址的数据单元，双字指令
MVDM	dmad MMR	dmad→DAR If (RC)≠0 Then (DAR 寻址的数据单元内容)→MMR，(DAR)+1→DAR Else (DAR 寻址的数据单元内容)→MMR，双字指令
MVDP	Smem, pmad	pmad→PAR If (RC)≠0 Then (Smem)→PAR 寻址的程序单元，(PAR)+1→PAR Else (Smem)→PAR 寻址的程序单元，双字指令

续表(九)

助记符指令形式		执行操作及描述
MVKD	dmad, Smem	dmad→DAR If (RC) ≠ 0 Then　(DAR 寻址的数据单元内)→Smem，(DAR)+1→DAR Else　(DAR 寻址的数据单元内)→Smem，双字指令
MVMD	MMR, dmad	dmad→EAR If (RC) ≠ 0 Then (MMR)→EAR 寻址的数据单元，(EAR) + 1→EAR Else (MMR)→EAR 寻址的数据单元，双字指令
MVMM	MMRx, MMRy	(MMRx)→MMRy，(MMRx/MMRy=AR0～AR7,SP)，单字指令
MVPD	pmad, Smem	pmad→PAR If (RC) ≠ 0 Then　(PAR 寻址的程序单元内容)→Smem，(PAR)+1→PAR Else　(PAR 寻址的程序单元内容)→Smem，双字指令
PORTR	PA, Smem	(PA)→Smem，PA 为 16 bit 外部 I/O 口地址，即从 I/O 口地址读入数据放入到数据单元中，双字指令
PORTW	Smem, PA	(Smem)→PA，PA 为 16 bit 外部 I/O 口地址，双字指令
READA	Smem	A→PAR If ((RC) ≠ 0)　(PAR 寻址的程序空间内容))→Smem (PAR)+1→PAR，(RC)–1→RC Else　(PAR 寻址的程序空间内容))→Smem，单字指令
WRITA	Smem	A→PAR If ((RC) ≠ 0) (Smem)→PAR 寻址的程序空间，(PAR)+1→PAR，(RC)–1→RC Else　(Smem)→PAR 寻址的程序空间，单字指令

在指令表中用到的非操作符符号的含义如下：

src	累加器 A 或 B，作源操作数
dst，dst_	累加器 A 或 B，作目的操作数，若有两个 dst，一个是 A，则另一个是 B，分别记为 dst 和 dst_
TS	T 暂存寄存器中 bit5～0 指定的移位量：－16≤TS≤31
ASM	ST1 寄存器中指定的累加器移位量：–16≤ASM≤15
Smem	16 bit 数据存储器操作数，单寻址，可以是直接寻址或间接寻址
Xmem,Ymem	16 bit 数据存储器操作数，用于双操作数指令或单操作数指令(Xmem)，间接寻址
Pmem	16 bit 程序存储器操作数
Lmem	32 bit 数据存储器操作数，长字寻址
dmad	16 bit 常量表示的数据空间地址
pmad	16 bit 常量表示的程序空间地址

expmad　23 bit 常量表示的程序空间地址，其中低 16 bit 将传给 PC，高 7 bit 传给 XPC

lk　16 bit 立即数

MMR　存储器映射寄存器

C　进位，状态寄存器 ST0 的 bit11

||　两条指令可并行执行

SHFT　4 位数：0≤SHFT≤15

SHIFT　5 位数：−16≤SHIFT≤15

C16　决定双字操作模式，C16＝0 为双精度(32 位)加/减，C16＝1 为两个 16 位数加/减

Sind　间接寻址操作数

BITC　4 位数，指定对哪一位测试，0≤BITC≤15

SBIT　4 位数，指定状态寄存器的位，0≤SBIT≤15

<<　移位操作，后跟正数是左移，负数为右移

TC　ST0 寄存器中的测试标志位

RC　循环计数值

TRN　转换寄存器，用于 Viterbi 译码

[D]　迟延跳转

指令表中的条件码如表 2.24 所示。所有与累加器 ACC 有关的条件码适用于 A 或 B 累加器。2 个或 3 个条件码可组合成复合条件，其组合规则为：

(1) 不同类可组合，例如 EQ 和 OV 可组合，TC、C、BIO 可组合。

(2) (1)中的不同类组合时，必须针对同一累加器。

(3) 不同组不能组合。

(4) 同组同类不能组合。

表 2.24　条 件 码

操 作 符	说　明	备　注
AEQ，BEQ	ACC＝0	组 1 类 A
ANEQ，BNEQ	ACC≠0	组 1 类 A
ALT，BLT	ACC＜0	组 1 类 A
ALEQ，BLEQ	ACC≤0	组 1 类 A
AGT，BGT	ACC＞0	组 1 类 A
AGEQ，BGEQ	ACC≥0	组 1 类 A
AOV，BOV	ACC 溢出	组 1 类 B
ANOV，BNOV	ACC 未溢出	组 1 类 B
TC	TC＝1	组 2 类 A
NTC	TC＝0	组 2 类 A
C	ALU 进位	组 2 类 B
NC	ALU 未进位	组 2 类 B
BIO	$\overline{\text{BIO}}$ 管脚为低	组 2 类 C
NBIO	$\overline{\text{BIO}}$ 管脚为高	组 2 类 C

单字指令并不一定是单周期指令，但一般来说单字指令比多字指令执行速度要快。

4．指令说明

1) 符号扩展

当数据(通常为 16 位)送入 40 位的算术逻辑单元 ALU 时，若 ST1 中的 SXM=0，则 ACC 的高位填 0；若 SXM=1，则高位符号扩展。

2) 溢出与饱和

当发生溢出时，OVA 或 OVB 位自动置 1。当 ST1 中的 OVM=1 时，正向溢出使累加器取饱和值(即 32 位最大正数)007FFFFFFFh，负向溢出取最大负饱和值 FF80000000h。SAT 指令直接使累加器饱和，与 OVM 无关。

3) 双 16 位加/减

将 ST1 中的 C16 置位，ALU 即可在单周期内同时进行 2 个 16 位数的加或减。这些指令包括 DADD、DADST、DRSUB、DSADT、DSUB 和 DSUBT。

4) 累加器存储

40 位累加器要用 3 个存储单元来存储值，或者根据需要移位存储，例如累加器 A 的值为 FF82345678h，则有如下存储方法：

```
STH    A，8，BUF1        ；存储 3456H
STH    A，-8，BUF2       ；存储 FF82H
STL    A，8，BUF3        ；存储 7800H
```

5) 累加器移位

累加器虽然是 40 位的，但移位时可能被当作只有低 32 位的寄存器，高 8 位填 0。如对于指令 ROR、ROL、ROLTC 来说，移位时符号位是否扩展、进位 C 是否参与移位等都比较复杂。

SFTA 是算术移位，移位数在-16～15 之间，是对整个 40 位操作；当右移时，若 SXM=1，高位符号扩展，若 SXM=0，高位填 0；左移时低位填 0。

SFTL 是逻辑移位，不受 SXM 影响，右移位时高位填 0，左移位时低位填 0。

SFTC 是条件移位，当 bit31 和 bit30 同号时才左移 1 位，以消除多余的符号位(归一化)。

ROL 为带进位 C 的循环左移 1 位，进位 C 移进累加器的 bit0，bit31 移到进位 C 中，累加器的最高 8 位(保护位)清 0。

ROR 为带进位 C 的循环右移 1 位，进位 C 移进累加器的 bit31，bit0 移到进位 C 中，最高 8 位(保护位)清 0。

ROLTC 为带 TC 位的循环左移 1 位，bit31 移到进位 C 中，TC 移到 bit0 中，最高 8 位(保护位)清 0。

6) 乘法器

乘法器有多种操作方式和结果处理方式：

两个 16 位无符号数相乘时，每个 16 位数前再加一位 0，形成 17 位数。

两个 16 位符号数相乘时，每个符号数都进行 1 位符号扩展形成 17 位数。

一个无符号数与一个符号数相乘，前者高位加 0，后者高位符号扩展，形成两个 17 位数。

ST1 中的 FRCT＝1 时，乘法器工作于小数方式，乘法结果左移 1 位以消去多余的符

号位。

乘法指令(MAC、MACA、MAS 等)如果带有后缀 R，就对乘法结果(或乘加结果)取整(加 2^{15}，传至累加器时，低 16 位清 0)。

7) CSSU 单元

CSSU 单元(即比较、选择、存储单元)用于 Viterbi 算法，指令为 CMPS。

8) 指数编码

指数编码可以将多余的符号位统计出来并用归一化指令消去。

```
例  EXP   A     ；将累加器 A 的冗余符号位数减 8(不计保护位)后，存入 T 寄存器
    NORM  A     ；对累加器 A 按 T 值进行移位以去掉多余的符号位，即归一化
```

9) 迟延跳转

迟延跳转可以将一条跳转指令的执行时间从 4 个周期减少为 2 个周期，但它使程序的可读性变差。迟延跳转相当于先执行跳转指令后续的一个双字节指令或两个单字节指令，然后才跳转。

10) 条件执行与流水线冲突

条件执行指令 XC 的效率比普通跳转指令高，它的操作原理为：

```
指令 1      ；此条指令决定 XC 中的条件
指令 2      ；对 XC 的条件无影响
指令 3      ；对 XC 的条件无影响
XC n，条件  ；n=1 或 2，条件满足时，执行后续 n 条指令
指令 5      ；条件满足时，执行此条指令，否则执行 NOP 指令
指令 6      ；条件满足且 n=2 时，执行此条指令，否则执行 NOP 指令
指令 7      ；条件是否满足都执行此条指令
```

上式中 n 指的是单字指令条数，若 XC 指令后是一条双字指令，则仅此一条指令被条件执行。

要注意 XC 指令的条件在前 2 条指令就已被确定。当条件不满足时，DSP 执行 n 条 NOP 指令，因此无论条件满足与否，这段程序体的执行时间是一样的(后续 n 条指令为单周期指令时)。

DSP 的指令流水线有时会冲突，下面的例子就存在流水线冲突。

```
例  STLM   A，AR0    ；对 MMR 寄存器 AR0 写
    LD     *AR0，B
```

第 2 条指令用 AR0 间接寻址时，AR0 值还没有按用户需要进行修改，为解决这一问题，应在两条指令中间插入 NOP，这样第 1 条指令的执行效果才能体现在第 2 条指令上。

流水线冲突问题在 C5000/C6000 系列 DSP 设计中非常重要，需要编程者仔细研究指令的流水执行序列，当发现冲突时，应插入 NOP 指令或重新安排指令顺序。如果程序是用 C 语言编写的，则 C 编译器生成的汇编指令不会发生流水线冲突。如果用汇编语言编程，特别是在用到 MMR 写操作时，需要考虑流水线的冲突问题，而在新推出的汇编工具中，汇编器会对 MMR 写操作冲突问题给出告警。编程者如果发现程序执行异常时，应考虑是否源于流水线冲突。

11) 并行指令

C5000 支持并行指令，但当并行的两条指令有相同的操作数时，将遵循先读后写原则，即当一个操作数既作为源操作数，又作为目的操作数时，源操作数为旧值(称为先读)，并行指令执行完毕后，此操作数(目的操作数)内容才被更新为新值(称为后写)。

12) 指令重复

为了高效率地循环执行指令，C5000 提供了单指令重复指令 RPT、RPTZ 和指令块重复指令 RPTB。它们在执行时省去了跳转引起的附加开销，节省了条件判断、流水线打断等占用的指令时间。在 RPT 或 RPTZ 执行期间，对 $\overline{\text{NMI}}$ 和所有可屏蔽中断都不响应。RPT 和 RPTZ 还能将乘法/累加和数据块传送等多周期指令变为单周期指令(第一次执行仍是多周期的)，这些指令包括 FIRS、MACD、MACP、MVDK、MVDM、MVDP、MVKD、MVMD、MVPD、READA 和 WRITA。其中单指令重复执行 READA、WRITA 时，还将初始地址由累加器 A 指定的程序空间地址在每次传送后自动加 1，用于搬移连续数据块。

然而有些指令不能用于单指令重复，这些指令包括 ADDM、ANDM、B[D]、BACC[D]、BANZ[D]、BC[D]、CALA[D]、CALL[D]、CC[D]、CMPR、DST、FB[D]、FBACC[D]、FCALA[D]、FCALL[D]、FRET[D]、FRETE[D]、IDLE、INTR、LD ARP、LD DP、MVMM、ORM、RC[D]、RESET、RET[D]、RETE[D]、RETF[D]、RND、RPT、RPTB[D]、RPTZ、RSBX、SSBX、TRAP、XC 和 XORM。

在执行 RPTB 指令前必须设置好 BRC 寄存器中的指令块循环执行次数。在指令块循环执行期间，可以响应中断。

当有多重循环时，可以按下述方法来安排循环以减少执行时间：

(1) 最内层循环用 RPT 或 RPTZ，使用 RC 寄存器，但只能用于单条指令循环。

(2) 次内层循环用 RPTB 指令，使用 BRC、RSA(块起始地址)和 REA(块终止地址)3 个寄存器。

(3) 其余层用 BANZ 指令，常用如下指令：BANZ 标号，*ARx-。该指令使 ARx 减 1，若 ARx 不为零，则跳转到循环体开始(标号)；若 ARx 为 0，结束此层循环，执行后续指令。

5. 指令执行时间优化

编写高效率的程序需要结合 DSP 的内部结构和特点，下面以 VC5402 为例介绍优化方法。

1) 片内多总线和片内访问

VC5402 片内有 4 套总线，每套都有独立的地址总线和数据总线，简称为 P、C、D、E 总线。

(1) P 总线用于取指令和立即操作数(立即操作数包括在指令码中)，还用于从程序空间向数据空间传数。

(2) C 和 D 总线用于从数据空间读操作数。片内双存取存储器(DARAM)被分块，可以在一个指令周期内从一块中读取两个操作数，并将数据通过 E 总线写入另一个块，一个 32 位长字的读相当于两套总线(C 和 D)同时读。

(3) E 总线用于向数据空间写数，也可向程序空间写数。

因此，如果合理安排操作数、系数、结果的存储器布局，则可以在一个指令周期内用 4 套总线完成 4 次存储器访问(3 个读和一个写)。例如，FIRS 指令利用 P、C、D 三总线可完

成 3 次读操作。

以上均针对 VC5402 片内存储器而言，若程序、数据放在片外，而 VC5402 仅有的一套外总线在一个指令周期内最多只能完成一次读操作(零等待)，或用 2 个指令周期完成一次写操作(零等待)，则这种多操作数指令要用多个指令周期才能完成。

VC5402 对存储器映射寄存器(MMR)的访问也要视情况而定：寻址 0000～001Fh 的寄存器为单周期；寻址 0020～005Fh 至少要 2 个周期。

2) 两个地址产生器

两个地址产生器支持双操作数的间接寻址。

```
例   MPY  *AR2+, *AR3+, A        ；双操作数间接寻址
```

这种双操作数间接寻址仅限于 AR2～AR5。

```
例   ST   A   *AR5
     || LD   *AR2+, B            ；并行操作
```

3) 长字指令

长字读指令为单周期，它利用 C、D 总线分别读取两个紧邻的 16 位数，作为一个 32 位数。长字写指令要用 E 总线操作 2 次，是双周期。

长字操作数的排列常用偶地址排列，即指令中给出的地址为偶地址，存高 16 位，后跟的奇地址存放低 16 位；奇地址排列则相反，指令中的地址为奇地址，存高 16 位。

参照前面的指令说明，可根据需要选用具体的指令以减少执行时间。直接寻址指令速度快；单字指令快于多字指令；单指令重复和指令块重复花费小；单指令重复可将一些多周期指令变为单周期指令。此外还有乘、加、减运算指令，如 MAC、MAS；双 16 位运算指令；并行运算指令；迟延跳转指令及循环寻址、位反序寻址等。

6．汇编源代码例子

下面利用 VC5402 实现一个 FIR 滤波器，FIR 滤波器的实现框图如图 2.3 所示。

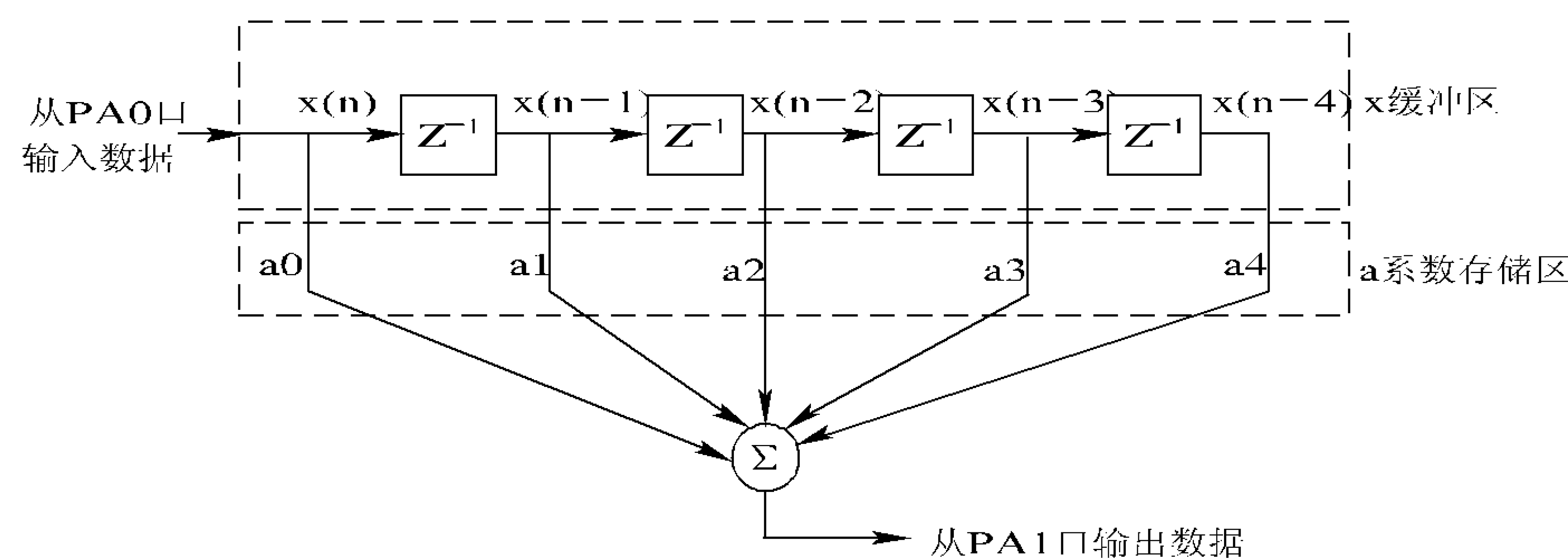


图 2.3　FIR 滤波器的实现框图

该滤波器的阶数为 5，数据从 PA0 I/O 口输入(利用 PORTR 指令)，滤波结果从 PA1 口输出(利用 PORTW 指令)。利用线形缓冲区 x 来存放滤波器的抽头值，最新数据放到缓冲区的顶端(最低地址)。当每一个新数据到来时，缓冲区中的值会向后(高地址)移动一个存储单元。存储区 a 用来存放滤波器系数。实现此 FIR 滤波器的源代码如下所示(只作为演示用，并不一定代表实际应用，用户可以自己编写出效率更高的应用程序)。

```
            .bss    x, 5                    ; 滤波器抽头缓冲区
            .bss    y, 1                    ; 滤波结果的暂存单元
            .data
a           .word   1, 2, 3, 4, 5           ; 滤波器系数 a0, a1, a2, a3, a4
PA0         .set    0                       ; 指定数据输入 I/O 口地址
PA1         .set    1                       ; 指定结果输出 I/O 口地址
            .text
start:      SSBX    1, FRCT                 ; 小数乘法
            RSBX    1, CPL                  ; CPL 清零，利用 DP 直接寻址
            STM     #x+4, AR1               ; AR1 指向 x(n- 4)
            STM     #a+4, AR2               ; AR2 指向 a4
            STM     #4, AR0                 ; 地址修正值 4→AR0
            LD      #x, DP                  ; 缓冲区的高 9 bit 地址放入到 DP 中
            PORTR   PA0， @x                ; 从 PA0 输入数据 x(n)
loop:       LD      *AR1- ， T              ; x(n- 4)→T
            MPY     *AR2- ,  A              ; a4*x(n- 4)→A
            LTD     *AR1-                   ; x(n- 3)→T， x(n- 3)→x(n- 4)
            MAC     *AR2- , A               ; A+a3*x(n- 3)→A
            LTD     *AR1-                   ; x(n- 2)→T， x(n- 2)→x(n- 3)
            MAC     *AR2- , A               ; A+a2*x(n- 2)→A
            LTD     *AR1-                   ; x(n- 1)→T， x(n- 1)→x(n- 2)
            MAC     *AR2- , A               ; A+a1*x(n- 1)→A
            LTD     *AR1                    ; x(n)→T， x(n)→x(n- 1)
            MAC     *AR2+0, A               ; A+a0*x(n)→A， AR2+AR0→AR2， AR2 重新
                                            ; 指向 a4
            STH     A, @y                   ; 暂存到 y 中
            PORTW   @y,  PA1                ; 从 PA1 口输出
            BD      loop                    ; 延迟跳转，重新开始接收下一个数据
            PORTR   PA0, *AR1+0             ; 输入 x(n)， AR1 重新指向 x(n- 4)
```

用户也可以利用 RPTZ、MACD 指令来提高程序的执行速度，但要求滤波器抽头缓冲区和滤波器系数分别放入数据空间和程序空间。

上面的 FIR 例子可以用如下的链接命令文件(fir.cmd)来配置存储器空间(VC5402)：

```
MEMORY
  {
    PAGE 0:   PROG: origin = 80h, length = 1F80h
    PAGE 1:   DATA: origin = 2000h, length = 2000h
  }
SECTIONS
    {
     .text > PROG    PAGE 0
     .data > DATA    PAGE 1
     .bss > DATA     PAGE1
    }
```

2.1.7 TMS320C5000 DSP 的 C/C++语言编程

C/C++编译器的操作方法在第 3 章关于集成开发环境 CCS 的介绍中再作说明，本节只对有关 TMS320C5000 的 C/C++语言编程的注意问题进行说明。

1. C 编译器的输出段

C 语言程序经 C 编译器编译后，自动输出如下代码和数据段：

.text：包含所有的可执行代码和编译器产生的常数，应放在程序空间中。

.cinit：包含用来初始化变量和常数的表，应放在程序空间中。

.pinit：包含全局结构体表，应放在程序空间中。

.switch：包含 switch 表，应放在程序空间中。

.const：包含字符串和用 const 定义的数据常数，应放在数据空间。

.bss：存放静态和全局变量，在加载过程中，加载程序会从.cinit 段中复制数据用来初始化这些静态和全局变量，应放在数据空间的 RAM 区。

.stack：系统堆栈存储空间，用于保护函数的返回地址，分配局部变量，在函数调用时传递参数，保护临时结果等。.stack 段的大小由链接器命令行的-stack 选项(在 CCS 一章中再介绍)来指定，默认值为 1024 字(16 bit)。程序代码不会检查堆栈是否溢出，因此用户必须计算好程序运行过程中用到的最大堆栈容量，用链接器命令行-stack 选项来分配足够的堆栈大小，以免溢出发生，导致不可预期的结果。程序利用堆栈指针 SP 来管理堆栈。.stack 段应放在数据空间的 RAM 区。

.sysmem：为动态存储空间分配保留空间，C/C++利用 malloc、calloc、realloc 函数来动态分配存储空间，此段应放在数据空间的 RAM 区。当然，如果 C/C++程序中没有动态分配存储空间，此段就不会产生。

需要注意的是，汇编程序中还会用到三个基本段(表 2.21 中)：.text、.bss 和 .data 及用户自己定义的段。当把汇编程序和 C/C++程序的目标代码链接时，汇编程序的.text 和.bss 段分别和 C/C++程序输出的.text 和.bss 段连接到一起，而汇编程序中的.data 和用户定义的段必须在链接命令文件中指定到存储空间中(当然，如果汇编程序中没有用到这些段就不用指定了)。C/C++程序利用.cinit 段来存放初始化表，汇编程序一定不能利用此段做其它目的。下面给出了一个包含 C/C++程序和汇编程序的链接命令文件(*.cmd)的例子(VC5402)。

```
/**************************************************/
/ 链接命令文件例子 lnk.cmd
/**************************************************/
-c                  /* ROM 自动初始化模式*/
-m example.map      /* 生成一个 map 文件 */
-o example.out      /* 输出可执行文件名*/
main.obj            /* 第一个 C/C++程序模块 */

sub.obj             /* 第二个 C/C++程序模块*/
asm.obj             /* 汇编程序模块*/
-l rts.lib          /* 运行时支持库 */
-l matrix.lib       /* 运算库 */

  MEMORY
    {
```

```
        PAGE 0 : PROG: origin = 80h, length = 1F80h
        PAGE 1 : DATA: origin = 2000h, length = 2000h
    }
SECTIONS
    {
    .text         >   PROG   PAGE 0
    .cinit        >   PROG   PAGE 0
    .switch       >   PROG   PAGE 0
    .bss          >   DATA   PAGE 1
    .const        >   DATA   PAGE 1
    .sysmem       >   DATA   PAGE 1
    .stack        >   DATA   PAGE 1
    .data         >   DATA   PAGE 1
    .usersect     >   DATA   PAGE 1
    }
```

2．数据类型

TMS320C5000 的 C/C++语言中定义的数据类型如表 2.25 所示，表中包括每一种数据类型的位长、表示方法和取值范围。

表 2.25　TMS320C5000 C/C++语言的数据类型定义

数据类型	位数	表示方法	数值范围
signed char	16	ASCII	- 32 768～32 767
char, unsigned char	16	ASCII	0～65 535
short, signed short, int, signed int	16	补码	- 32 768～32 767
unsigned short,unsigned int	16	原码	0～65 535
long, signed long	32	补码	- 214 748 364 8～214 748 364 7
unsigned long	32	原码	0～429 496 729 5
float ,double, long double	32	IEEE 32 bit 浮点格式	1.175494e- 38～3.40282346e+38
enum	16	补码	- 32 768～327 67
pointers	16	原码	0～0xFFFF

在为变量或常数分配存储空间时，一定要特别注意这些数据类型的位数及其数值范围，如不小心可能会导致错误结果。

3．变量初始化

在程序开始运行前，需要对一些变量进行初始化。标准 C 语言要求：如果没有特别指定初始化值，那么静态和全局变量全部预初始化为 0。这种初始化是在加载过程中执行的，然而加载过程依靠指定的目标系统环境，并不一定保证把静态和全局变量初始化为 0，这一点用户应该特别注意。用户处理程序应该避免把静态和全局变量假定为具有 0 初始化值来使用，用户也可以给静态和全局变量特别指定初始化为 0，或在链接命令文件中把 .bss 段填充为 0，即

```
SECTIONS
  {
    ...
    .bss: fill = 0x00;
    ...
  }
```

4．在 C/C++程序中调用汇编函数

在 C 程序中调用汇编函数需要注意以下几点原则：函数名有何要求及如何声明，输入/

输出参数如何传递，汇编程序中用到的哪些寄存器需要保护，C 程序中用到的存储器和汇编程序用到的存储器如何分配，即链接命令文件如何修改。以上原则应用于所有 DSP 的 C 程序和汇编程序的链接，下面针对 TMS320C5000 的编译器，介绍如何满足上述原则。

1) 函数名要求

在 C/C++主程序的开头首先对要调用的汇编函数声明为外部函数(例如：extern void sub();)；在汇编程序中此函数名前必须有一下画线(例如：_sub)，并用.global 伪指令声明为全局符号(例如：.global _sub)。对于不被 C 程序访问的其它符号前不要加下划线。

2) 参数传递

C/C++程序在调用汇编函数时，把第 1 个输入参数放入到累加器 A 中，把其它的输入参数按逆序压入到堆栈中，即最右边的输入参数放入到最高地址处，而第 2 个输入参数放入到最低地址处，然后再把函数的返回地址(即函数返回后继续执行的程序的地址)压入到堆栈中，PC 跳转到被调用函数处。

被调用汇编函数可以继续利用堆栈来保存寄存器和其局部变量，图 2.4 为函数调用过程中堆栈的使用情况。如果被调用函数有返回值，其返回值也要保存在累加器 A 中。汇编函数一般用来作需要大运算量的处理(例如作 FFT 处理)，在 C 程序中对数据分配空间(例如：int a[1024];)，把输入数据存储区和输出结果存储区的首地址传递给汇编处理函数(FFT(a);)，经过汇编程序处理后，结果保存在输出存储区中，此时汇编函数不需要返回值。

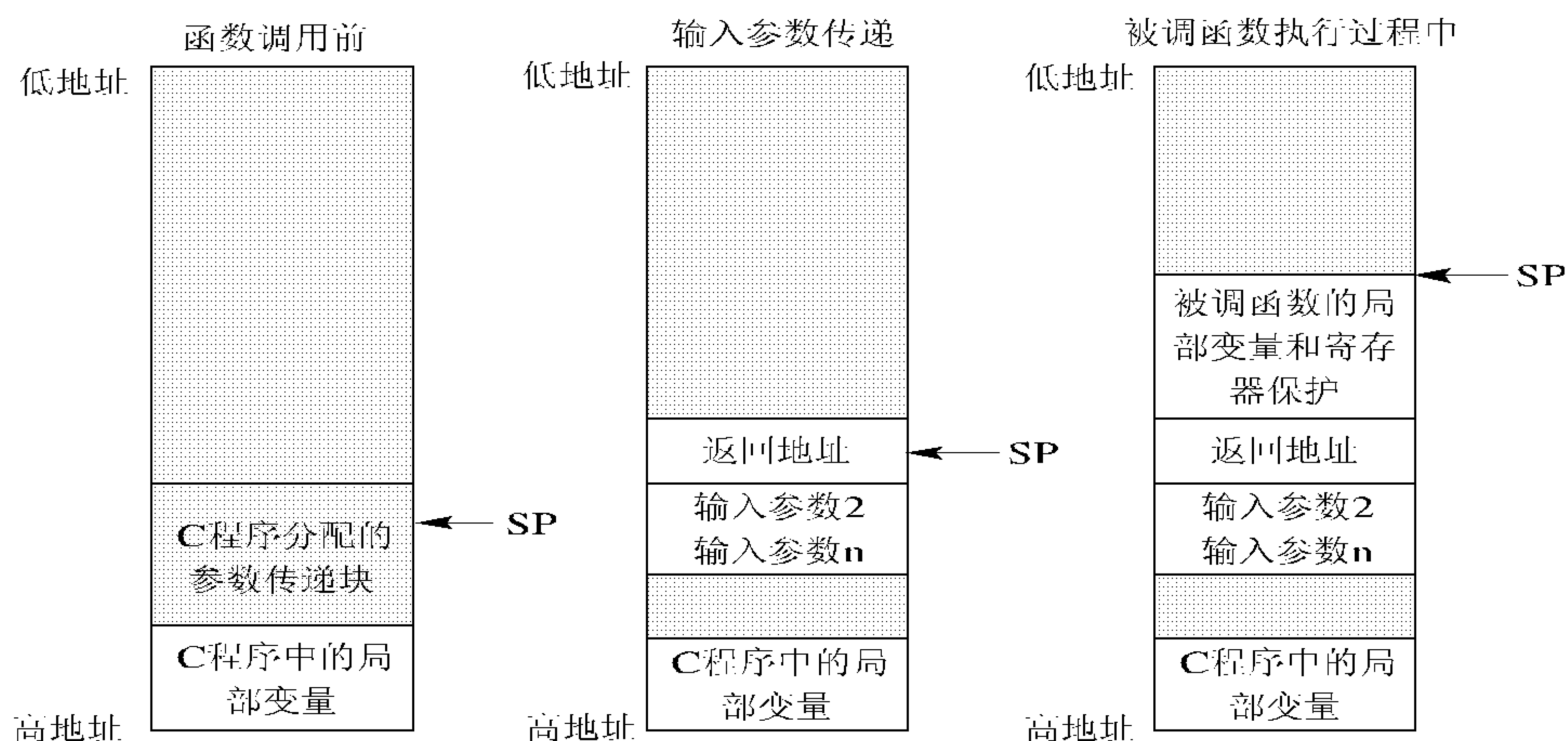


图 2.4 函数调用过程中堆栈的使用情况

3) 寄存器保护

如果被调用的汇编函数中使用了 AR1、AR6 和 AR7 寄存器，则必须在汇编函数的开始处把这些寄存器压入堆栈保护，而在汇编函数返回前应从堆栈弹出这些寄存器，恢复其原值。对于汇编函数中使用的其它寄存器，C/C++程序在调用函数前后自动保护，用户不需要考虑。

如果压入栈的所有内容在函数返回前都弹出，SP 就不用保护，否则 SP 也要保护。

如果是中断程序则应保护所有使用的寄存器。

4) 存储区分配及链接命令文件修改，在前面已作了介绍，这里不再重述。

下面给出一个在 C 程序中调用汇编函数的例子。

```
/*************C 程序*****************/          /*********汇编程序***********/
extern int asmsub(int, int *);  /*外部函数声明*/          .global _asmsub
int a[1024];               /* 定义全局数组变量*/          .text
main()                                           _asmsub:  PSHM AR6    ; 保护 AR6
 {                                                         PSHM AR7    ; 保护 AR7
  int i;                                                      ...      ; 函数主体
  ⋮                                                        POPM AR7    ; 恢复 AR7
  asmsub(a);               /* 调用汇编函数 */              POPM AR6    ; 恢复 AR6
  ⋮                                                        RET         ; 返回，栈顶内容→PC
 }
```

5．在 C/C++程序中插入汇编行

有时希望直接对 DSP 硬件操作，例如对寄存器操作、设置中断、设置定时器等，而 C/C++语言做不到这一点，可以通过在 C/C++程序中插入一条或多条汇编指令来实现。在 C/C++程序中插入汇编指令的格式如下：

```
asm("汇编指令");    /*在 C/C++程序的当前位置插入汇编指令*/
```

在 C/C++程序中插入汇编指令时，一定要特别注意不要破坏 C/C++程序的环境，例如对某一寄存器修改时，就可能会导致错误的结果。

6．从 C/C++程序中访问汇编程序中的全局变量

如果汇编程序中的变量在.bss 段中定义，则访问它的方法很简单，只要在汇编程序中将此变量名前加一下画线，并用.global 声明为全局变量，并在 C/C++程序中，把此变量声明为外部变量，就可以直接访问了。

如果汇编程序中的变量不在.bss 段中，例如在用户利用.usect 定义的段中，则在 C/C++程序中需要利用一个指针来访问此变量。

下面的例子给出了如何从 C/C++程序中访问汇编程序中的全局变量和常数。

```
/***********C 程序*************/                   /*************汇编程序**************/
extern int var;          /* 声明外部变量 */          _table_size .set 10000
extern float sine[];     /* 声明外部数组变量 */      .global   _table_size
extern int table_size;   /*声明外部常数 */           .bss _var,1        ; 定义变量
#define TABLE_SIZE   ((int) (&table_size))           .global _var       ; 声明为全局变量
                                                     .global _sine      ; 声明为全局变量
                                                     .sect "sine_tab"   ; 定义一个用户存储段
main()                                               _sine:           ; 在用户段中存放一个数据表
{                                                    .float 0.0
 int i;                                              .float 0.015987
 float f ;                                           .float 0.022145
 float *sine_p = sine; /* 定义指针指向 sine */          ⋮
 for (i=0; i< TABLE_SIZE; i++)
  f = sine_p[i];        /* 利用指针访问外部变量 */
  var = 1;              /* 直接访问外部变量 */
 }
```

7．从汇编程序中访问 C/C++程序中的全局和静态变量

从汇编程序中访问 C/C++程序中的全局和静态变量的方法是：在 C/C++程序中将变量

声明为全局或静态变量，然后在汇编程序中把此变量也声明为全局符号，但变量名前应加一下画线。下例给出了如何从汇编程序中访问 C/C++程序中的全局和静态变量。

```
                                                .global   _a
int    a[1024];        /*定义外部数组*/          .global   _var
static    var;         /*定义静态变量*/          STH    A, @_var    ; 保存 A 的值
main()                                          STM    #_a, AR1    ; AR1 指向 a 数组
{
}
```

8. C/C++编译器提供的运行时支持库

TMS320C5000 C/C++编译器提供如下运行时支持库：

- ts.lib——ISO 标准的运行时支持目标代码库。
- rts.src——rts.lib 的源代码库。

rts.lib 库包含如下内容：标准的 C/C++运行时支持函数、浮点算术函数、系统启动程序(_c_int00)、允许 C/C++程序访问的特定指令的函数或宏等。如果用户程序中用到这些运行时支持函数，就必须把 rts.lib 库链接到用户目标代码中。

9. C/C++程序初始化

C/C++编译器会在 C/C++主程序 main( )之前加入一个 C 初始化程序模块：_c_int00，它为 C/C++程序设置运行环境。_c_int00 函数包含在 rts.lib 库中，在链接 rts.lib 库和用户目标代码时，利用 -c 或 -cr 命令行选项，_c_int00 会自动加入到 main( )程序之前。_c_int00 程序完成以下内容：

- 为系统定义一个名为 .stack 的堆栈，并设置堆栈指针和帧指针。
- 将.cinit 内容拷贝到 .bss 段，对全局和静态变量初始化。
- 调用 main( )开始的 C 程序。

2.2　TMS320C6000 DSP 的内部功能结构及源代码开发

2.2.1　TMS320C6000 DSP 的功能和结构特点

TI 公司推出的 TMS320C6000 系列 DSP 是一种高性能 DSP 芯片。定点 DSP 包括 TMS320C62xx 和 TMS320C64xx 系列，浮点 DSP 包括 TMS320C67xx 系列。这三种 DSP 系列都采用高性能、先进的超长指令字(VLIW)结构，这种结构使 8 个功能单元并行执行，在单个时钟周期内可同时执行 8 条指令。C62xx 的所有定点指令都对 C64xx 和 C67xx 有效，同时 C64xx 和 C67xx 又各自具有自己的一些特定指令。

C62xx/C67xx 的 CPU 核中包含 32 个 32 bit 通用寄存器和 8 个功能单元，而 C64xx 的 CPU 核中具有 64 个 32 bit 通用寄存器和 8 个功能单元。

C6000 DSP 的共同特点如下：

- 具有 8 个功能单元的 VLIW 结构，这 8 个功能单元包括 2 个乘法器和 6 个 ALU 单元。
- 8 条 32 bit 指令组成一个指令包，这 8 条指令可以并行执行、串行执行或部分串行执行。

• 所有指令都可以条件执行。
• 提供 8/16 /32 bit 数据的存储器支持。
• 支持 40 bit 扩展精度定点算术运算。
• 对主要的算术操作提供饱和和归一化支持。
• 支持位操作。
• 具有片内程序和数据存储器，容量从 512 Kb 到 7 Mb 不等。
• 具有多种外设资源，包括：DMA、主机口、扩展总线、串口、定时器、JTAG 仿真口等。
• C62xx 的处理速度可达到 1600 MIPS，C64xx 的处理速度可达到 9000 MIPS，C67xx 的浮点运算速度可达到 1GFLOPS。

C67xx 附加的特点包括：

• 支持 32 bit(单精度)和 64 bit(双精度)IEEE 浮点操作。
• 支持 32 bit×32 bit 整数乘法，输出 32 bit 或 64 bit 结果。
• 4 个浮点/定点 ALU、2 个定点 ALU 和 2 个浮点/定点乘法器。

C64xx 附加的特点包括：

• 每个乘法器在单周期内可以完成 2 个 16 bit×16 bit 或 4 个 8 bit×8 bit 乘法运算。
• 4 个 8 bit 和 2 个 16 bit 指令扩展。
• 支持非 32 bit 和 64 bit 边界(non-aligned)的存储器访问。
• 提供专门的通信指定指令。
• 提供位计数和旋转的硬件支持。

C6000 的内部结构如图 2.5 所示(以 C620x/C670x 为例)。C6000 中各系列芯片的内部结构基本相同，只是片内外设资源及片内存储器的容量有所不同。

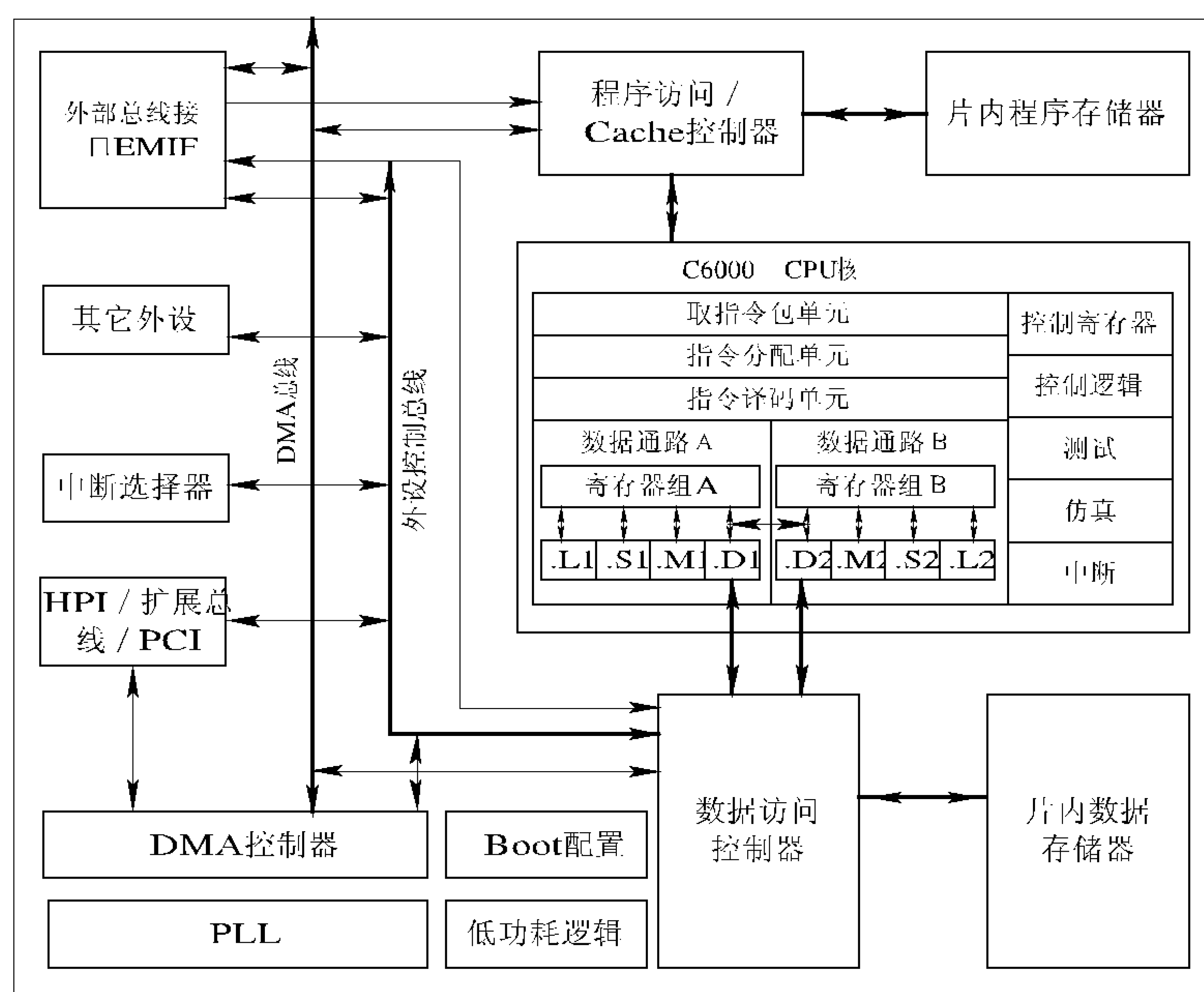


图 2.5　C6000 DSP 的内部结构

由图 2.5 可以看出，C62xx/C67xx/C64xx DSP 的内部结构由三个主要部分组成：CPU 核、内部存储器和片内外设资源。

2.2.2　CPU 核

CPU 核是 DSP 的运算和控制中心，它包括如下几部分：

• 程序取指单元、指令分配单元、指令译码单元；

• 两个数据通路 A 和 B，每个通路中有 4 个功能单元(.L，.S，.M，.D)和由 16 个 32 bit 寄存器(C64xx 为 32 个 32 bit 寄存器)组成的寄存器组；

• 控制寄存器；

• 控制逻辑；

• 测试、仿真和中断逻辑。

程序取指单元、指令分配单元和指令译码单元能够在每个 CPU 时钟周期内向 8 个功能单元移交 8 条 32 bit 指令(每个功能单元接收一条指令)。这些指令的处理在两个数据通路 A 和 B 中进行，每个数据通路包括 4 个功能单元(.L，.S，.M 和.D)和 16 个 32 bit 通用寄存器(C64xx 为 32 个 32 bit 通用寄存器)。功能单元执行逻辑、位移、乘法、加法和数据寻址等操作，除取指令和存指令之外的所有指令均对寄存器产生影响。两个数据寻址单元(.D1 和.D2)专门负责寄存器与存储器之间的数据传递。每个数据通路的 4 个功能单元有单一的数据总线连接到另一侧的寄存器组上，以便功能单元能够访问另一侧寄存器组中的操作数。控制寄存器用来对各种 DSP 操作进行配置和控制。

1．通用寄存器组

在 C6000 数据通路中有两个通用寄存器组 A 和 B。对于 C62xx/C67xx DSP，每一个寄存器组包含 16 个 32 bit 通用寄存器(组 A 包含 A0～A15，组 B 包含 B0～B15)。对于 C64xx DSP，每一个寄存器组包含 32 个 32 bit 通用寄存器(组 A 包含 A0～A31，组 B 包含 B0～B31)。这些通用寄存器用于数据、数据地址指针。寄存器 A1、A2、B0、B1 和 B2 可用于条件寄存器，寄存器 A4～A7 和 B4～B7 可用于循环寻址。

C62xx/C67xx 通用寄存器支持打包的 16 bit /32 bit /40 bit 定点和 64 bit 浮点数据，而 C64xx 的通用寄存器还支持打包的 8 bit 和 64 bit 定点数据。40 bit 和 64 bit 的数据需要用两个寄存器(组成一个寄存器对)来保存。对于 40 bit 数据，数据的低 32 位放在偶寄存器内，剩余的高 8 位放在比此偶寄存器序号大 1 的寄存器(即奇寄存器)的低 8 位内；而 64 bit 数据则占满整个奇寄存器的 32 位。在汇编语言中，寄存器对在两个寄存器之间加一冒号，且奇寄存器放在前面。表 2.26 中列出了 C62xx/ C67xx/ C64xx 中用于存放 40 bit 和 64 bit 数据的所有寄存器对。

表 2. 26　C6000 的 40 bit/64 bit 寄存器对

C62xx/C67xx/ C64xx		C64xx	
寄存器组 A	寄存器组 B	寄存器组 A	寄存器组 B
A1：A0	B1：B0	A17：A16	B17：B16
A3：A2	B3：B2	A19：A18	B19：B18
A5：A4	B5：B4	A21：A20	B21：B20
A7：A6	B7：B6	A23：A22	B23：B22
A9：A8	B9：B8	A25：A24	B25：B24
A11：A10	B11：B10	A27：A26	B27：B26
A13：A12	B13：B12	A29：A28	B29：B28
A15：A14	B15：B14	A31：A30	B31：B30

2. 功能单元

C6000 数据通路中的 8 个功能单元分成 2 组，每组 4 个功能单元。一个数据通路中的功能单元与另一个数据通路中的相应功能单元几乎相同。表 2.27 中列出了这些功能单元的操作描述。C64xx 增加了一些对 5 bit、8 bit 和 16 bit 数据操作的指令，它们在表 2.27 中以黑体表示。

表 2.27　C6000 功能单元的操作描述

功能单元	定点操作	浮点操作
.L 单元 (.L1,.L2)	32/40 bit 算术和比较操作 32 bit 逻辑操作 对 32 bit 中最左边 1 或 0 的位数计数 32 bit 和 40 bit 计数 字节移动 数据打包/展开 5 bit 常数产生 双 16 bit 算术操作 四 8 bit 算术操作 双 16 bit　min/max 操作 四 8 bit　min/max 操作	算术操作 DP(双精度) → SP(单精度)转换 INT(整型) → DP 转换 INT → SP 转换
.S 单元 (.S1,.S2)	32 bit 算术操作 32/40 bit 移位和 32 bit 位操作 32 bit 逻辑操作 跳转 常数产生 寄存器与控制寄存器数据传递(仅.S2) 字节移动 数据打包/展开 双 16 bit 比较操作 四 8 bit 比较操作 双 16 bit 移位操作 双 16 bit 饱和算术操作 四 8 bit 饱和算术操作	比较 倒数和倒数平方根操作 绝对值操作 SP → DP 转换操作
.M 单元 (.M1,.M2)	16×16 bit 乘法操作 16×32 bit 乘法操作 双 16×16 bit 乘法操作 四 8×8 bit 乘法操作 带加/减的双 16×16 bit 乘法操作 带加法的四 8×8 bit 乘法操作 位扩展 位交错/去交错存取 变量移位操作 旋转 有限域乘法	32×32 bit 定点乘法操作 浮点乘法操作
.D 单元 (.D1,.D2)	32 bit 加、减、线性和循环寻址计算 5 bit 常数偏移量存取 15 bit 常数偏移量存取(仅.D2) 5 bit 常数偏移量双字存取 存取非边界字和双字 5 bit 常数产生 32 bit 逻辑操作	5 bit 常数偏移量双字存储器读

注：表中的黑体表示 C64xx 附加的操作。

每个功能单元都有自己的写入通用寄存器的写口，1结尾的所有功能单元(如.L1)写入寄存器组A，2结尾的所有功能单元写入寄存器组B。每个功能单元都有2个32 bit源操作数src1和src2的读口。为了40 bit长型操作数的访问，4个功能单元(.L1,.L2,.S1和.S2)分别另外配有额外的8 bit写口和读口(L1和S1共用，L2和S2共用)。由于C64xx的每一乘法器能够返回64 bit结果，因此添加了另一个从乘法器向寄存器组的写口。

3．寄存器组交叉通路

每个功能单元可以直接对其所在的数据通路中的寄存器组进行读/写操作。寄存器组通过交叉通路1X和2X与另一侧的功能单元相连，交叉通路允许一侧数据通路的功能单元可以访问另一侧寄存器组中的32 bit操作数，1X交叉通路允许数据通路A的功能单元从寄存器组B中读它的源操作数，2X交叉通路允许数据通路B的功能单元从寄存器组A中读它的源操作数。

C62xx/C67xx的.M1、.M2、.S1、.S2单元的src2源操作数(src1表示第1个源操作数，src2表示第2个源操作数)和.L1、.L2单元的src1、src2源操作数都可以选择交叉通路的源操作数。C64xx的.M1、.M2、.S1、.S2、.D1、.D2单元的src2源操作数和.L1、.L2单元的src1、src2源操作数都可以选择交叉通路的源操作数。

C62xx/C67xx的每个数据通路中只能有一个功能单元在每周期内访问交叉通路，而C64xx的每个数据通路中可以有2个功能单元同时访问交叉通路。

4．存储器读写通路

C62xx有两个32 bit读通路可把数据从存储器读到寄存器中，寄存器组A的读通道为LD1，寄存器组B的读通道为LD2。C67xx另有两个32 bit读通道，允许LDDW指令同时读取2个32 bit数据到寄存器组A和2个32 bit数据到寄存器组B。C62xx/C67xx另有两个32 bit写通路ST1和ST2，用于把寄存器中的值写入到存储器中。

C64xx支持双字读写，因此C64xx有四个数据读通路和四个数据写通路。

5．数据地址通路

数据地址通路DA1和DA2分别连接到.D1和.D2功能单元上，允许任一通路产生的数据地址支持寄存器到存储器的读写操作。

6．控制寄存组

控制寄存器组用来配置和控制DSP的各种操作。只有功能单元.S2可对控制寄存器组进行读写操作，通过指令MVC访问每个控制寄存器。表2.28列出了控制寄存器组中的控制寄存器以及对每个控制寄存器的描述。

表2.28　C6000控制寄存器

控制寄存器缩写	控制寄存器名称	描　述
AMR	寻址模式寄存器	指定是否使用线性或循环寻址和循环缓冲区的长度
CSR	控制状态寄存器	包括全局中断使能位、cache 控制位和其它各种控制和状态
IFR	中断标志寄存器	指示中断挂起状态
ISR	中断设置寄存器	允许手工设置挂起的中断
ICR	中断清除寄存器	允许手工清除挂起的中断
IER	中断使能寄存器	允许使能/禁止个别中断
ISTP	中断服务表指针	指向中断服务表的开始地址
IRP	中断返回指针	保存从可屏蔽中断返回时的地址
NRP	不可屏蔽中断返回指针	保存从不可屏蔽中断返回时的地址
PCE1	E1节拍程序计数器	保存在E1流水线阶段执行的取指令包地址

1) 寻址模式寄存器(AMR)

C6000 的所有通用寄存器中，只有 8 个寄存器 A4～A7 和 B4～B7 可以执行循环寻址，其寻址模式由 AMR 控制，AMR 的位定义如图 2.6 所示。在寻址模式寄存器 AMR 中，对 A4～A7 和 B4～B7 中的每一寄存器都有 2 bit 位段指定地址修改模式：线性寻址或循环寻址。对于循环寻址，这 2 bit 位段同时也指定哪个 BK(BK0 或 BK1)位段指定循环缓冲区的长度。表 2.29 列出了这 2 bit 模式选择位段的编码定义。

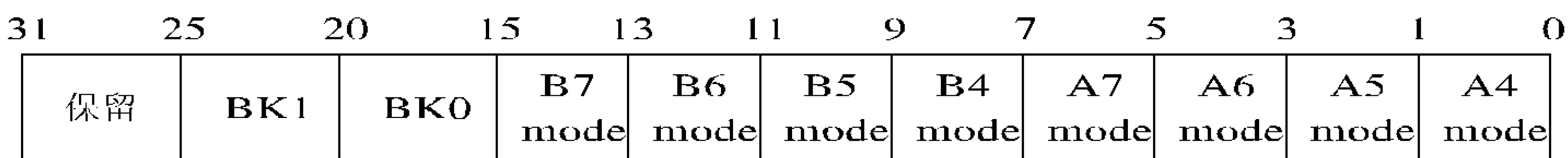


图 2.6　寻址模式寄存器(AMR)的位定义

表 2.29　AMR 的循环寻址模式位段的编码

模式(Mode)	描　述
00	线性寻址(复位缺省)
01	循环寻址，由 BK0 位段指定缓冲区长度
10	循环寻址，由 BK1 位段指定缓冲区长度
11	保留

BK0 或 BK1 位段的 5 位数值用于计算循环缓冲区的长度，长度与 BK0 或 BK1 的 5 位数值 N 的关系为：长度＝2^{N+1}。

2) 控制状态寄存器(CSR)

控制状态寄存器包含控制和状态位。表 2.30 中列出了 CSR 各位段的定义。对于 EN、PWRD、PCC 和 DCC 位段，需要查看相关数据手册以确定芯片是否支持这些位段的控制选项。

表 2.30　控制状态寄存器的位定义

位	名　称	功　能
31～24	CPU ID	CPU ID(识别号)，指定芯片的型号
23～16	REV ID	修订版号
15～10	PWRD	控制低功耗模式，该值读时总为零
9	SAT	饱和位，当功能单元执行一个饱和操作时被设置，饱和位只能靠 MVC 指令清除。当清除和设置同时发生时，设置优先于清除。饱和位在饱和发生后一个周期被设置
8	EN	1=little endian, 0=big endian
7～5	PCC	程序 cache 控制模式
4～2	DCC	数据 cache 控制模式
1	PGIE	当一个中断发生时，此位保存先前的 GIE(全局中断使能位)
0	GIE	全局可屏蔽中断使能位，1=使能，0=禁止除复位和不可屏蔽中断之外的所有中断

3) E1 节拍程序计数器(PCE1)

此寄存器用来保存在流水线中处于 E1 节拍的执行包的 32 位地址(关于 C6000 DSP 的流水线操作比较复杂，本书不作详细介绍，感兴趣的读者请查阅用户手册)。

7．TMS320C67xx 控制寄存器扩展

C67xx 还另外提供了三个配置寄存器以支持浮点操作。这些寄存器为.L、.S 和.M 功能单元指定所希望的浮点舍入方式，它们也用来对以下情况进行告警：源操作数 src1 和 src2 是无效数 NAN 或非归一化数；结果上溢出、下溢出、不准确、无穷大或无效；执行除以零操作；用 NAN 源操作数进行比较。表 2.31 列出了 C67xx 扩展的三个配置寄存器。当条件指令的条件为假时，这些寄存器的 OVER、UNDER、INEX、INVAL、DENn、NANn、INFO、UNORD 和 DIV0 位不被修改。

表 2.31　TMS320C67xx 扩展的配置寄存器

寄存器缩写	寄存器名称	描　　述
FADCR	浮点加法配置寄存器	指出.L 单元的溢出方式、舍入方式、NaNs 及其它例外
FAUCR	浮点辅助配置寄存器	指出.S 单元的溢出方式、舍入方式、NaNs 及其它例外
FMCR	浮点乘法配置寄存器	指出.M 单元的溢出方式、舍入方式、NaNs 及其它例外

1) 浮点加法配置寄存器(FADCR)

FADCR 分成两段：一段指定.L1 功能单元的位段(0～15 bit)，另一段指定.L2 功能单元的位段(16～31 bit)。表 2.32 列出了 FADCR 的位定义。

表 2.32　浮点加法配置寄存器的位定义

位	名　称	功　　能
31～27 / 15～11	Reserved	保留
26～25/10～9	RMode .L2/L1	00：舍入到最接近的浮点数 01：舍入为 0 10：舍入为无穷大 11：舍入为负无穷大
24/8	UNDER .L2/L1	结果下溢出时置 1
23/7	INEX .L2/L1	如果结果超出指数范围和精度界限，与期望的计数结果不同时置位；从不与 INVAL 同时置位
22/6	OVER .L2/L1	结果上溢出时置位
21/5	INFO .L2/L1	结果为无穷大时置位
20/4	INVAL .L2/L1	当有符号的 NaN 是源操作数，或在浮点到整型转换中 NaN 是一个源操作数，或无穷大减无穷大时置位
19/3	DEN2 .L2/L1	src2 是一个非归一化数时置位
18/2	DEN1 .L2/L1	src1 是一个非归一化数时置位
17/1	NAN2 .L2/L1	src2 是 NaN 时置位
16/0	NAN1 .L2/L1	src1 是 NaN 时置位

2) 浮点辅助配置寄存器(FAUCR)

FAUCR 分成两段(0～15 bit 和 16～31 bit)，分别服务于.S1 和.S2 功能单元。表 2.33 列出了 FAUCR 的位定义。

表 2.33　浮点辅助配置寄存器的位定义

位	名　称	功　　能
31～27/15～11	—	保留
26/10	DIV0.S2/S1	当 0 作为求倒数操作的源操作数时置位
25/9	UNORD.S2/S1	当 NaN 作为比较操作的源操作数时置位
24/8	UNDER.S2/S1	结果下溢出时置位
23/7	INEX.S2/S1	当结果超出指数的范围和精度界限，计算结果与期望的结果不符时置位；不与 INVAL 同时置位
22/6	OVER.S2/S1	结果上溢出时置位
21/5	INFO.S2/S1	当结果是无穷大时置位
20/4	INVAL.S2/S1	当有符号的 NaN 是源操作数，或在浮点数向整数的转换中 NaN 是源操作数，或无穷大减无穷大时置位
19/3	DEN2.S2/S1	当 src2 是非归一化数时置位
18/2	DEN1.S2/S1	当 src1 是非归一化数时置位
17/1	NAN2.S2/S1	当 src2 是 NaN 时置位
16/0	NAN1.S2/S1	当 src1 是 NaN 时置位

3) 浮点乘法配置寄存器(FMCR)

FMCR 分成两段(0～15 bit 和 16～31 bit)，分别服务于.M1 和.M2 功能单元。表 2.34 列出了 FMCR 的位定义。

表 2.34　浮点乘法配置寄存器的位定义

位	名　称	功　　能
31～27/15～11	Reserved	保留
26～25/10～9	RMode .M2/M1	00：舍入到最接近的浮点数 01：舍入为 0 10：舍入为无穷大 11：舍入为负无穷大
24/8	UNDER .M2/M1	结果下溢出时置位
23/7	INEX .M2/M1	如果结果超出指数范围和精度界限，与预期的计数结果不同时置位；从不与 INVAL 同时置位
22/6	OVER .M2/M1	结果上溢出时置位
21/5	INFO .M2/M1	结果为无穷大时置位
20/4	INVAL .M2/M1	当有符号的 NaN 是源操作数，或在浮点到整型转换中 NaN 是一个源操作数，或无穷大减无穷大时置位
19/3	DEN2 .M2/M1	src2 是一个非归一化数时置位
18/2	DEN1 .M2/M1	src1 是一个非归一化数时置位
17/1	NAN2 .M2/M1	src2 是 NaN 时置位
16/0	NAN1 .M2/M1	src1 是 NaN 时置位

8．TMS320C64xx 控制寄存器扩展

TMS320C64xx 中增加了一个控制寄存器：伽罗瓦域多项式产生函数寄存器，简称 GFPGFR。表 2.35 为 GFPGFR 的位定义。

表 2. 35　伽罗瓦域多项式产生函数寄存器的位定义

位	名　称	功　能
31～27	Reserved	保留
26～24	SIZE	域大小
23～8	Reserved	保留
7～0	POLY	多项式产生器

2.2.3　存储器组织

1．存储器映射

C6201/C6202(B)/C6203(B)/C6204/C6205/C6701 有两种存储器映射方式：MAP0 和 MAP1，这两种映射方式的不同之处在于 MAP0 的外部存储器位于首地址为 0 的存储空间，而 MAP1 的片内存储器位于 0 地址空间。表 2.36 列出了 C620x/C670x 的存储器映射地址。

表 2. 36　C620x/C670x 的存储器映射

地址范围(Hex)	长度字节	存储器块描述	
		MAP0	MAP1
0000 0000～0000 FFFF ～0003 FFFF ～0005 FFFF	64 K 256 K 384 K	外部存储器接口 CE0	片内程序 RAM(C6201/C6204/C6205/C6701) (C6202(B)) (C6203(B))
0001 0000～003F FFFF 0004 0000～ 0006 0000～	4 M-64 K 4 M-256 K 4 M-384 K	外部存储器接口 CE0	保留 (C6201/C6204/C6205/C6701) (C6202(B)) (C6203(B))
0040 0000～00FF FFFF	12 M	外部存储器接口 CE0	外部存储器接口 CE0
0100 0000～013F FFFF	4 M	外部存储器接口 CE1	外部存储器接口 CE0
0140 0000～0140 FFFF ～0143 FFFF ～0145 FFFF	64 K 256 K 284 K	片内程序 RAM	外部存储器 CE1(C6201/C6204/C6205/C6701) (C6202(B)) (C6203(B))
0141 0000～017F FFFF 0144 0000～ 0146 0000～	4 M-64 K 4 M-256 K 4 M-384 K	保留	外部存储器接 CE1(C6201/C6204/C6205/C6701) (C6202(B)) (C6203(B))
0180 0000～0183 FFFF	256 K	内部外设总线 EMIF 寄存器	
0184 0000～0187 FFFF	256 K	内部外设总线 DMA 控制器寄存器	

续表

地址范围(Hex)	长度字节	存储器块描述	
		MAP0	MAP1
0188 0000～018B FFFF	256 K	内部外设总线 HPI(C6201/ C6701) XBUS 寄存器(C6204/ C6202(B)/ C6203(B)) 保留(C6205)	
018C 0000～018F FFFF	256 K	内部外设总线 McBSP 0 寄存器	
0190 0000～0193 FFFF	256 K	内部外设总线 McBSP 1 寄存器	
0194 0000～0197 FFFF	256 K	内部外设总线 Timr 0 寄存器	
0198 0000～019B FFFF	256 K	内部外设总线 Timr 1 寄存器	
019C 0000～019F FFFF ～019C 01FF 019C 0200～019F FFFF	256 K 512 256 K- 512	内部外设总线中断选择寄存器(C6201/C6204/C6205/C6701) (C6202(B)/C6203(B)) 内部外设总线 powr-down 寄存器(C6202(B)/C6203(B))	
01A0 0000～01A3 FFFF	256 K	保留	
01A4 0000～01A8 FFFF ～01A7 FFFF	320 K 256 K	内部外设总线 PCI 寄存器 (C6205) 保留 (C6201/C6204/C6701) 内部外设总线 McBSP2 寄存器(C6202(B)/C6203(B))	
01A9 0000～01FF FFFF 01A8 0000～	6 M- 576 K 5.5 M	保留 (C6201/C6204/C6205/C6701) (C6202(B)/C6203(B))	
0200 0000～02FF FFFF	16 M	外部存储器接口 CE2	
0300 0000～03FF FFFF	16 M	外部存储器接口 CE3	
0400 0000～3FFF FFFF	1 G- 64 M	保留	
4000 0000～4FFF FFFF	256 M	扩展总线 XEC0 (C6204/C6202(B)/C6203(B)) 保留(C6201/C6205/C6701)	
5000 0000～5FFF FFFF	256 M	扩展总线 XEC1 (C6204/C6202(B)/C6203(B)) 保留(C6201/C6205/C6701)	
6000 0000～6FFF FFFF	256 M	扩展总线 XEC2 (C6204/C6202(B)/C6203(B)) 保留(C6201/C6205/C6701)	
7000 0000～7FFF FFFF	256 M	扩展总线 XEC3 (C6204/C6202(B)/C6203(B)) 保留(C6201/C6205/C6701)	
8000 0000～8000 FFFF ～8001 FFFF ～8007 FFFF	64 K 128 K 512 K	片内数据 RAM (C6201/C6204/C6205/C6701) (C6202(B)) (C6203(B))	
8001 0000～FFFF FFFF 8002 0000～ 8008 0000～	2 G- 64 K 2 G- 128 K 2 G- 512 K	保留(C6201/C6204/C6205/C6701) (C6202(B)) (C6203(B))	

C621x/C671x 只有一种存储器映射方式，内部存储器总是位于 0 地址空间处。内部存储器既可以作为程序存储器也可以作为数据存储器。表 2.37 为 C621x/C671x 存储器映射。

表 2.37　C621x/C671x 存储器映射

地址范围(Hex)	长度(字节)	存储块描述
0000 0000～0000 FFFF	64 K	内部 RAM(L2)
0001 0000～017F FFFF	24 M- 64 K	保留
0180 0000～0183 FFFF	256 K	内部配置总线 EMIF 寄存器
0184 0000～0187 FFFF	256 K	内部配置总线 L2 控制寄存器
0188 0000～018B FFFF	256 K	内部配置总线 HPI 寄存器
018C 0000～018F FFFF	256 K	内部配置总线 McBSP 0 寄存器
0190 0000～0193 FFFF	256 K	内部配置总线 McBSP 1 寄存器
0194 0000～0197 FFFF	256 K	内部配置总线 timer 0 寄存器
0198 0000～019B FFFF	256 K	内部配置总线 timer 1 寄存器
019C 0000～019F FFFF	256 K	内部配置总线中断选择寄存器
01A0 0000～01A3 FFFF	256 K	内部配置总线 EDMA RAM 和寄存器
01A4 0000～01FF FFFF	6 M- 256 K	保留
0200 0000～0200 0033	52	QDMA 寄存器
0200 0034～2FFF FFFF	736 M- 52	保留
3000 0000～3FFF FFFF	256 M	McBSP0/1 数据
4000 0000～7FFF FFFF	1 G	保留
8000 0000～8FFF FFFF	256 M	外部存储器接口 CE0
9000 0000～9FFF FFFF	256 M	外部存储器接口 CE1
A000 0000～AFFF FFFF	256 M	外部存储器接口 CE2
B000 0000～BFFF FFFF	256 M	外部存储器接口 CE3
C000 0000～FFFF FFFF	1 G	保留

C64xx 类似于 C621x，也只有一种存储器映射方式，内部存储器总是位于 0 地址空间处。内部存储器既可以作为程序存储器也可以作为数据存储器。表 2.38 为 C64xx 存储器映射。

表 2.38　C64xx 存储器映射

地址范围(Hex)	长度(字节)	存储块描述
0000 0000～000F FFFF	1 M	内部 RAM(L2)
0010 0000～017F FFFF	23 M	保留
0180 0000～0183 FFFF	256 K	内部配置总线 EMIFA 寄存器
0184 0000～0187 FFFF	256 K	内部配置总线 L2 寄存器
0188 0000～018B FFFF	256 K	内部配置总线 HPI 寄存器
018C 0000～018F FFFF	256 K	内部配置总线 McBSP0 寄存器
0190 0000～0193 FFFF	256 K	内部配置总线 McBSP1 寄存器
0194 0000～0197 FFFF	256 K	内部配置总线 timer0 寄存器
0198 0000～019B FFFF	256 K	内部配置总线 timer1 寄存器
019C 0000～019F FFFF	256 K	内部配置总线中断选择寄存器

续表

地址范围(Hex)	长度(字节)	存储块描述
01A0 0000～01A3 FFFF	256 K	内部配置总线 EDMA RAM 和寄存器
01A4 0000～01A7 FFFF	256 K	内部配置总线 McBSP2 寄存器
01A8 0000～01AB FFFF	256 K	内部配置总线 EMIFB 寄存器
01AC 0000～01AF FFFF	256 K	内部配置总线 timer2 寄存器
01B0 0000～01B3 FFFF	256 K	内部配置总线 GPIO 寄存器
01B4 0000～01B7 FFFF	256 K	内部配置总线 UTOPIA 寄存器(C6415) 保留(C6414)
01B8 0000～01BF FFFF	512 K	保留
01C0 0000～01C3 FFFF	256 K	内部配置总线 PCI 寄存器(C6415) 保留(C6414)
01C4 0000～01FF FFFF	4 M- 256 K	保留
0200 0000～0200 0033	52	QDMA 寄存器
0200 0034～2FFF FFFF	736 M- 52	保留
3000 0000～33FF FFFF	64 M	McBSP0 数据
3400 0000～37FF FFFF	64 M	McBSP1 数据
3800 0000～3BFF FFFF	64 M	McBSP2 数据
3C00 0000～3FFF FFFF	64 M	UTOPIA　列队(C6415) 保留(C6414)
4000 0000～5FFF FFFF	512 M	保留
6000 0000～63FF FFFF	64 M	EMIFB　CE0
6400 0000～67FF FFFF	64 M	EMIFB　CE1
6800 0000～6BFF FFFF	64 M	EMIFB　CE2
6C00 0000～6FFF FFFF	64 M	EMIFB　CE3
7000 0000～7FFF FFFF	256 K	保留
8000 0000～8FFF FFFF	256 K	EMIFA　CE0
9000 0000～9FFF FFFF	256 K	EMIFA　CE1
A000 0000～AFFF FFFF	256 K	EMIFA　CE2
B000 0000～BFFF FFFF	256 K	EMIFA　CE3
C000 0000～CFFF FFFF	1 G	保留

C6201/6701 的存储器映射方式通过 BOOTMODE[4:0]管脚设置，同时 BOOTMODE[4:0]也决定了复位后芯片的引导方式。C6202(B)/C6203(B)/C6204 没有这 5 个管脚，取而代之的是将扩展总线的 XD[4:0]映射为 BOOTMODE[4:0]，这些管脚可通过电阻上拉或下拉设置引导方式和存储器映射方式。而 C6205 把 EMIF 数据总线的 ED[4:0]映射为 BOOTMODE[4:0]，BOOTMODE[4:0]决定引导方式和存储器映射方式。C621x/C671x/C64xx 只有一种存储器映射方式，引导方式只需要 2 位进行设置，C621x/C671x 利用主机口的 HD[4:3]，而 C64xx 利用 EMIFB 地址总线的 BEA[19:18]。

2．片内存储器

不同芯片的片内存储器的容量和组织结构有所不同。C620x/C670x 芯片具有分开的片内程序和数据存储器，片内程序存储器既可以作为程序存储区也可以作为程序 cache。C6202(B)/C6203(B)有点不同，它们提供了另一片内存储块，只用于程序存储区，不作为 cache 使用。片内程序存储区是否用于程序 cache 由控制状态寄存器(CSR)的 PCC 位段设置，PCC 位段的定义由表 2.39 给出。

表 2.39　内部程序存储区的模式选择

片内程序存储区模式	PCC 值	描　　述
映射存储器	000	cache 禁止(芯片复位时默认)
cache 使能	010	cache 能被访问和更新
cache 冻结	011	cache 能被访问但不更新
cache 旁路	100	cache 不能被访问也不能更新，确保外部存储器的程序被读取
—	其它	保留

当内部程序存储器设置为 cache 模式(PCC＝010，011 或 100)时，C6201/C6204/C6205/6701 所有的内部程序存储器都作为 cache 使用，此时程序指令应放入外部存储器中。而 C6202(B)/C6203(B)的内部程序存储器只是部分作为 cache 使用，而另一部分仍然作为程序 RAM。表 2.40 列出了在 cache 模式下，C6202(B)/C6203(B)中仍然作为程序 RAM 的存储器的地址范围。

表 2.40　C6202(B)/C6203(B)在 cache 模式下的内部程序 RAM 的地址

芯　片	长　度	MAP0(Hex)	MAP1(Hex)
C6202(B)	128 K	0140 0000～0141 FFFF	0000 0000～0001 FFFF
C6203(B)	256 K	0140 0000～0143 FFFF	0000 0000～0003 FFFF

C6201/C6204/C6205 的片内数据 RAM 为 64 K 字节，它被分成两个 32 K 字节的存储块，分别位于地址空间 8000 0000h～8000 7FFFh 和 8000 8000h～8000 FFFFh 内，每个存储块又由四个 4 K 16 位的 bank 组成。CPU 的数据通路 A 和 B 能够同时访问两个相邻 bank 的 16 bit 数据。按这种存储器结构，CPU 和 DMA 在一个时钟周期内可以最多访问三个 32 bit 数据，只要 CPU 的数据通路 A、B 和 DMA 分别访问不同的 bank 对。

C6701 片内的 64 K 数据 RAM 也分成两个 32 K 字节的存储块，分别位于地址空间 8000 0000h～8000 7FFFh 和 8000 8000h～8000 FFFFh 内。与 C6201/C6204/C6205 所不同的是，C6701 的每个存储块由八个 2 K 16 bit 的 bank 组成。CPU 的数据通路 A 和 B 也能够同时访问两个相邻 bank 的 16 bit 数据。按这种存储器结构，每时钟周期内可以访问的最大数据量为：两个 64 bit 数据(CPU 访问，LDDW 指令)和一个 32 bit 数据(DMA 访问)。

C6202(B)片内的数据 RAM 为 128 K，分成两个 64 K 字节的存储块，分别位于 8000 0000h～8000 FFFFh 和 8001 0000h～8001 FFFFh 地址空间内。每个存储块由四个 8 K 16 bit 的 bank 组成。其组织结构及其 CPU、DMA 数据访问方式与 C6201/C6204/C6205 的类似。

C6203(B)片内的数据 RAM 为 512 K，分成两个 256 K 字节的存储块，分别位于

8000 0000h～8003 FFFFh 和 8004 0000h～8007 FFFFh 地址空间内。每个存储块由四个 32 K 16 bit 的 bank 组成。其组织结构及其 CPU、DMA 数据访问方式与 C6201/C6204/C6205 的类似。

图 2.7 为 C6201/C6204/C6205 片内数据 RAM 的组织结构(只显示了第一个数据块)。C6701、C6202(B)和 C6303(B)片内数据 RAM 的组织结构与之类似。

Bank 0	Bank 1	Bank 2	Bank3
80000000 80000001	80000002 80000003	80000004 80000005	80000006 80000007
80000008 80000009	8000000A 8000000B	8000000C 8000000D	8000000E 8000000F
⋮ ⋮	⋮ ⋮	⋮ ⋮	⋮ ⋮
80007FF0 80007FF1	80007FF2 80007FF3	80007FF4 80007FF5	80007FF6 80007FF7
80007FF8 80007FF9	80007FFA 80007FFB	80007FFC 80007FFD	80007FFE 80007FFF

图 2.7　C6201/C6204/C6205 片内数据 RAM 的组织结构

C621x/C671x/C64xx 的内部存储器具有两级结构：第一级只作为高速缓冲区 cache 使用，包括程序 cache(L1P)和数据 cache(L1D)，不能设置为映射存储器；第二级(L2)可以作为映射存储器，用来存储程序和数据，L2 也可以部分地配置为 cache 使用。L2 的配置模式受 cache 配置寄存器 CCFG 的控制，表 2.41 为 C621x/C671x 和 C64xx 的 L2 存储器配置模式。

表 2.41　C621x/C671x 和 C64xx 的 L2 存储器配置模式

L2 模式	C621x/C671x			C64xx		
	cache 范围(Hex)	映射存储器范围(Hex)	映射存储器长度(字节)	cache 范围(Hex)	映射存储器范围(Hex)	映射存储器长度(字节)
000	—	0000 0000 ～0000 FFFF	64 K	—	0000 0000 ～000F FFFF	1 M
001	0000 C000 ～0000 FFF	0000 0000 ～0000 BFFF	48 K	000F 8000 ～000F FFFF	0000 0000 ～000F 7FFF	992 K
010	0000 8000 ～0000 FFFF	0000 0000 ～00007FFF	32 K	000F 0000 ～000F FFFF	0000 0000 ～000E FFFF	960 K
011	0000 4000 ～0000 FFFF	0000 0000 ～0000 3FFF	16 K	000E 0000 ～000F FFFF	0000 0000 ～000D FFFF	896 K
111	0000 0000 ～0000 FFFF	—	0	000C 0000 ～000F FFFF	0000 0000 ～000B FFFF	768 K
其它	保留					

C621x/C671x 的程序 cache(L1P)为 4 K 字节，64 字节线长。C621x/C671x 的数据 cache(L1D)也为 4 K 字节，分成两路，每路 32 字节线长。

C64xx 的程序 cache(L1P)不支持 cache 冻结和旁路，长度为 16 K 字节，32 字节线长。C64xx 的数据 cache(L1D)不支持 cache 冻结和旁路，长度为 16 K 字节，64 字节线长。

C6000 存储器的数据组织形式有以下几种：

- 双字(64 bit)：双字数据在存储器中存放地址的最低 3 位必须为零，C64xx 不受此限制；
- 字(32 bit)：32 bit 数据在存储器中存放地址的最低 2 位必须为零，C64xx 不受此限制；

• 半字(16 bit)：16 bit 数据在存储器中存放地址的最低位必须为零。16 bit 数据访问要求这 16 bit 数据在一个 16 bit bank 内；

• 字节(8 bit)：可以位于任意地址空间上。

最后将 C6000 的片内存储器配置总结于表 2.42 中。

表 2.42 C6000 的片内存储器配置

芯 片	片内存储器总容量	程序存储器(字节)	数据存储器(字节)	统一存储器(字节)
C6201/C6204/ C6205/C6701	128 K	64 K(map/cache)	64 K(map)	无
C6202(B)	384 K	128 K(map) 128 K(map/cache)	128 K(map)	无
C6203(B)	896 K	256 K(map) 128 K(map/cache)	512 K(map)	无
C621x/C671x	72 K	4 K(L1P)	4 K(L1D)	64 K(L2)
C64xx	1 M+32 K	16 K(L1P)	16 K(L1D)	1 M(L2)

2.2.4 中断

1. 中断类型和中断信号

TMS320C6000 的 CPU 中有三种中断类型：复位中断(RESET)、不可屏蔽中断(NMI)和可屏蔽中断(INT4～INT15)。这些中断的优先级顺序如表 2.43 所示。

表 2.43 C6000 中的断优先级

优先级	中断名
最高级	RESET
	NMI
	INT4
	INT5
	INT6
	INT7
	INT8
	INT9
	INT10
	INT11
	INT12
	INT13
	INT14
最低级	INT15

复位中断具有最高优先级，相应信号为 $\overline{\text{RESET}}$ 信号，不可屏蔽中断具有第二优先级，相应信号为 NMI 信号。可屏蔽中断具有最低优先级，其信号为 INT4～INT15。$\overline{\text{RESET}}$、NMI 和部分 INT4～INT15 信号映射到 C6000 芯片的管脚上，有些 INT4～INT15 信号被片内外设所使用，有些则不可用，或者在软件控制下使用。使用时请查看相关数据手册。

1) 复位($\overline{\text{RESET}}$)中断

复位是最高级别中断，用来停止 CPU 工作，并使之返回到一个已知状态。

2) 不可屏蔽中断(NMI)

NMI 具有第二优先级，通常用来向 CPU 发出一系列硬件问题的警报，如掉电。为实现不可屏蔽中断处理，中断使能寄存器中的不可屏蔽中断使能位(NMIE)必须置 1。NMIE 在复位时被清零，以防止复位被打断。由于在一个 NMI 发生时，NMIE 被清零，这样就阻止了另一个 NMI 的处理。NMIE 不可人为地清零，但可以设置允许 NMI 嵌套。在 NMIE 为 0 时，所有可屏蔽中断(INT4～INT15)也被禁止。

3) 可屏蔽中断(INT4～INT15)

C6000 CPU 中有 12 个可屏蔽中断，这些中断的优先级别比 NMI 和复位中断低，可连接到外部设备、片内外设，也可软件控制或者不用。假设一个可屏蔽中断不发生在跳转的延迟期间内，则必须同时满足下列条件时可屏蔽中断才被处理：

- 控制状态寄存器(CSR)中的全局可屏蔽中断使能位 GIE 置位；
- 中断使能寄存器(IER)中的 NMIE 位置位；
- IER 中的相应可屏蔽中断使能位 INT4～INT15 置位；
- 中断标志寄存器(IFR)的相应位置位，且在 IFR 中没有更高优先级别的中断标志(IF)位置位。

4) 中断应答(IACK 和 INUMx)

IACK 和 INUMx 信号通知 C6000 外部硬件，一个中断已经发生且正在进行处理。IACK 信号指出 CPU 已经开始处理一个中断。INUMx 信号(INUM3～INUM0)指出正在处理的是哪一个中断(即 IFR 中标志位的位置)。

例　INUM3＝0　(MSB)
　　INUM2＝1
　　INUM1＝1
　　INUM0＝1　(LSB)

这些信号一起提供了 4 位数据 0111，指出 INT7 正在处理。

2．中断服务表(IST)

中断服务表 IST(Interrupt Service Table)即中断矢量表，是包含中断服务取指包的一个地址表。IST 包含 16 个连续取指包，每个中断服务取指包都含 8 条指令。

图 2.8 的例子中示出了中断矢量表。由于每个取指包都有 8 条 32 bit 指令字(或 32 个字节)，因此表中的地址以 32 个字节(即 20h)增长。

1) 中断服务取指包(ISFP)

ISFP(Interrupt Service Fetch Packet)是用于中断服务的取指包。当中断服务程序很小时，可以把它放在一个单独的取指包(FP)内，如图 2.8(a)中的例子(INT6 的 ISFP)所示。为了中断结束后能够返回主程序，FP 中包含一条跳转到中断返回指针所指向地址的指令(B IRP)，接着是一条 NOP5 指令，这条指令使跳转指令有效地进入流水线的执行级，若没有这条指令，CPU 将会在跳转完成之前执行 IST 中下面的 5 个执行包(跳转指令有 5 个延迟间隙，我们在后文中再详细介绍)。

如果中断服务程序太长而不能放在单个 FP 内时，就需要跳转到另外的中断服务程序的地址上。图 2.8(b)中的例子为 INT4 的中断服务程序，由于程序太长，一部分程序放在地址 1234h 开始的存储器内，因此在 INT4 的 ISFP 内有一条跳转到 1234h 的跳转指令。因为跳转指令有 5 个延迟间隙，所以 B 1234h 放在 ISFP 中间。另外，即使当 1220h～1230h 与 1234h 为并行指令(后文中再详细介绍)时，CPU 也不执行 1220h～1230h 内的指令(关于跳转指令将在第 2.2.6 节中再详细介绍)。

2) 中断服务表指针寄存器(ISTP)

中断服务表指针寄存器 ISTP(Interrupt Service Table Pointer)用于指定中断服务取指包的地址。ISTP 中的位段 ISTB 指定 IST 的基地址，另一位段 HPEINT 指定这一中断取指包在

IST 中的位置。表 2.44 给出了 ISTP 的位定义。

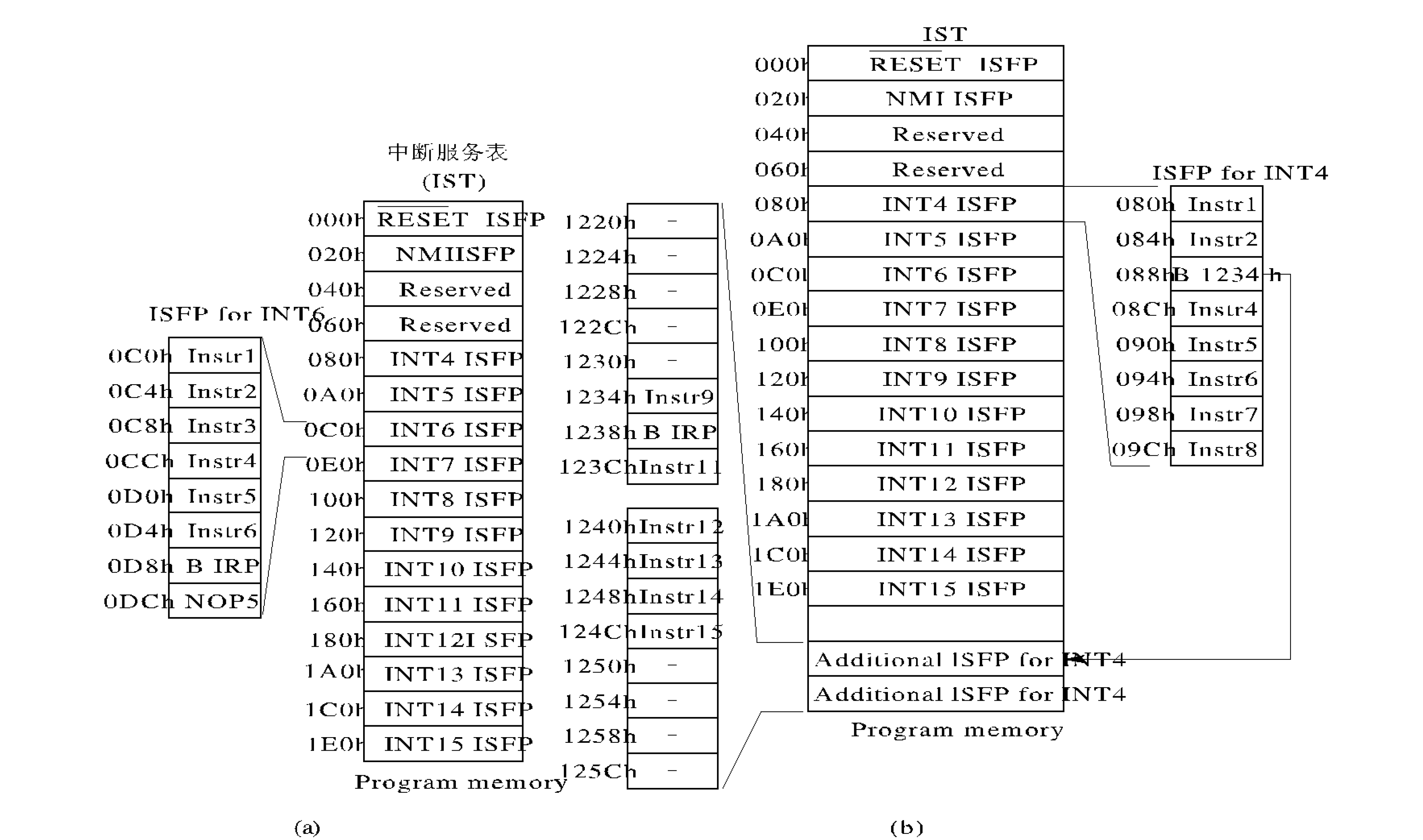


图 2.8　中断服务程序例子

表 2.44　中断服务表指针寄存器的位定义

位	名　称	描　　述
4～0	—	置 0(取指包必须位于 8 字(32 字节)边界上)
9～5	HPEINT	该位段给出已使能且挂起的最高优先级中断的序号(对应 IFR 中的位置)。因此，利用 ISTP 可以手工跳转到已使能的最高优先级中断上。如果没有中断挂起和使能，HPEINT 的值为 00000b
31～10	ISTB	IST 的基地址。复位时置 0，因此复位时 IST 必须放在 0 地址处。复位后，可对 ISTB 写入新的数值以重新定位 IST。如果重新定位，第一个 ISFP(对应 RESET 中断)从不执行，因为复位会使 ISTB 置 0

复位取指包必须放在 0 地址空间处。IST 的位置由中断服务表基值(ISTB)指定。下例为中断服务表的重新定位。

例　中断服务表的重新定位。

(1) 定位 IST 到 800h：

① 将地址为 0h～200h 的 IST 内容移到地址 800h～A00h 中；

② 写 800h 到 ISTP 寄存器：

```
MVK 800h, A2
MVK A2, ISTP
ISTP=800h=1000 0000 0000 b
```

(2) ISTP 引导 CPU 到重新定位后的 ISFP：

假设：IFR=BBC0h=101 $\bar{1}$ 10 $\bar{1}$ 1 1100 0000b

IER=1230h= 000 $\bar{1}$ 00 $\bar{1}$ 0 0011 0000b

显然两个使能且挂起的中断为 INT9 和 INT12，IFR 中的 1 表示挂起的中断，IER 中的 1 表示被使能的中断，因为 INT9 的优先级别高于 INT12 的，因此 HPEINT 的编码应为 INT9 的值 01001b，而 HPEINT 为 ISTP 中的 bit9～bit5，故 ISTP＝1001 0010 0000b=920h=INT9 的地址。

3．中断控制寄存器

C6000 芯片有 8 个中断控制寄存器，它们总结于表 2.45 中。

表 2.45　中断控制寄存器

缩写	名　称	描　　述
CSR	控制状态寄存器	全局使能或禁止可屏蔽中断
IER	中断使能寄存器	使能或禁止单个中断
IFR	中断标志寄存器	指示已挂起的中断
ISR	中断标志设置寄存器	允许手工设置 IFR 中的标志位
ICR	中断标志清除寄存器	允许手工清除 IFR 中的标志位
ISTP	中断服务表指针	重新定位中断服务表
NRP	不可屏蔽中断返回指针	包含从不可屏蔽中断返回的地址，该中断返回通过指令 B NRP 完成
IRP	可屏蔽中断返回指针	包含从可屏蔽中断返回的地址，该中断返回通过指令 B IRP 完成

1) 控制状态寄存器(CSR)

CSR 中的 GIE 和 PGIE 用于控制中断。CSR 的其它位段服务于其它目的，已在前面介绍过。

全局中断使能位 GIE(bit0)用来使能(置位)或禁止(清零)所有的可屏蔽中断。CSR 的 PGIE(bit1)用来保存先前的 GIE，即在可屏蔽中断期间 PGIE 保存 GIE 的值，而 GIE 被清零，从而防止另一个可屏蔽中断的发生。当从中断返回时，通过 B IRP 指令，使 PGIE 的值重新返回到 GIE 中。

2) 中断使能寄存器(IER)

IER 的位定义如图 2.9 所示。通过对 IER 中相应中断位的置位/清零来使能/禁止此中断。

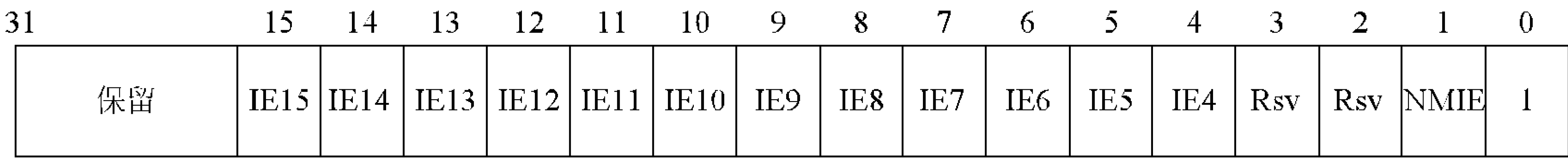


图 2.9　中断使能寄存器的位定义

IER 的 bit 0 对应于复位中断，该位可读(读作 1)不可写，由于 bit 0 总为 1，所以复位中断总被使能。NMIE＝0 时，禁止所有非复位中断。NIME＝1 时，GIE 和相应的 IER 位一起使能 INT15～INT4 中断。对 NMIE 写 0 无效，只有复位中断或 NMI 发生时 NMIE 才清零，NMIE 的置位靠 B NRP 指令或写 1 指令完成。

例　使能个别中断 INT9。

```
MVK     200h, B1        ; 置 bit 9
MVC     IER, B0         ; 获取 IER 的值
OR      B1, B0, B0      ; 只置位 bit 9，其它位不变
MVC     B0,IER          ; 置位 IER 中的 bit 9
```

例 禁止个别中断 INT9。

```
MVK     FDFFh, B1       ; 把 bit 9 清零
MVC     IER, B0         ; 获取 IER 的值
AND     B1, B0, B0      ; 只清零 bit 9，其它位不变
MVC     B0, IER         ; 把 IER 中的 bit 9 清零
```

3) 中断标志、中断标志设置和清除寄存器(IFR，ISR，ICR)

中断标志寄存器(IFR)指示 INT4～INT15 和 NMI 的挂起情况。当一个中断发生时，IFR 中的相应中断标志位置位，否则为 0。如果需要查看中断状态，则利用 MVC 指令读取 IFR 的值。IFR 的中断标志位与 IER 中的位一一对应。

中断标志设置寄存器(ISR)和中断标志清除寄存器(ICR)可以用来手工设置和清除 IFR 中的可屏蔽中断位，其位定义与 IFR 的可屏蔽中断标志位一一对应。对 ISR 的 IS4～IS15 位写 1 则会使 IFR 中对应的中断标志位置位；对 ICR 的 IC4～IC15 位写 1 则会使 IFR 中对应的标志位清 0。对 ISR 和 ICR 的任何位写 0 无效。设置或清除 ISR 和 ICR 的任何位都不影响 NMI 和复位。新进来的中断会忽略任何对 ICR 的写入。另外，写入 ISR 或 ICR(靠 MVC 指令)有一个时钟周期延迟间隙。当同时对 ICR 和 ISR 的同一位写入 1 时，ICR 被忽略，ISR 写入。

例 置位中断 INT6 和读 IFR。

```
MVK     40h, B3
MVC     B3, ISR
NOP
MVC     IFR, B4
```

例 清除中断 INT6 和读 IFR。

```
MVK     40h, B3
MVC     B3, ICR
NOP
MVC     IFR, B4
```

4) 不可屏蔽中断返回指针寄存器(NRP)

NRP 包含从不可屏蔽中断返回时的指针，该指针引导 CPU 返回到原来程序执行的正确位置。当 NMI 服务完成时，为返回到被中断的原程序中，在中断服务程序末尾必须安排一条跳转到 NRP 的指令(即 B NRP)。下例代码给出了怎样从 NMI 中断服务程序中返回。

例 从 NMI 返回程序。

```
B   NRP         ; 返回，置位 NMIE
NOP   5         ; 5 个时钟周期的延迟间隙
```

5) 可屏蔽中断返回指针寄存器(IRP)

可屏蔽中断返回指针寄存器的功能与 NRP 类似，包含从可屏蔽中断返回时的指针，该

指针引导 CPU 返回到原来程序执行的正确位置。下例的代码给出了如何从可屏蔽中断服务程序中返回。

例　从可屏蔽中断返回代码。

```
B   IRP        ; 返回，PGIE 中的值复制到 GIE 中
NOP  5         ; 5 个时钟周期的延迟间隙
```

4. 中断性能和编程考虑

1) 一般性能

• 总开销：C62xx/C64xx 芯片所有 CPU 中断的总开销是 7(周期 6~12)个时钟周期，C67xx 则是 9(周期 6～14)个时钟周期。该期间没有新指令进入 E1 流水节拍。

• 反应时间：C62xx/C64xx 和 C67xx 的中断反应时间分别是 11 个周期和 13 个周期(复位中断为 21 个周期)，即从中断激活到执行中断服务程序需要 11(C62xx/C64xx)和 13(C67xx)个时钟周期。

• 两次中断的最小时间间隔：两次可识别中断的最小间隔是 2 个时钟周期，而两次中断处理间隔则取决于中断服务所需要的时间和是否使能嵌套中断。

2) 流水线与中断的相互影响

因为取指包的串行或并行编码不影响流水线中的 DC(指令译码)节拍及后来的各节拍，所以代码并行与中断不存在冲突。但有三个操作或条件影响中断或被中断影响。

• 跳转：任何执行包在 n 至 n+4 时钟周期间包含跳转或者处在跳转延迟期间时，则非复位中断被延迟；

• 存储器阻塞：因为存储器阻塞本身延长了 CPU 周期，所以存储器阻塞延迟了中断处理；

• 多周期 NOPs 指令：当发生中断时，多周期指令 NOPs(包括 IDLE)操作同其它指令一样。但有一个例外，就是当一个中断执行时多周期指令 NOPs 刚好处在其第一个周期。在这种情况下，下一个执行包的地址存放在 NRP 或者 IRP 中，这就阻止了返回到被中断的 NOPs 或 IDLE 指令处。

3) 单分配编程

当系统中有中断过程时，就要考虑寄存器的分配形式。单分配是可中断的，多分配是不可中断的，否则会出现不可预料的结果。我们知道，当中断发生时，所有进入 E1 节拍的指令允许完成整个执行过程，而其它指令被废除，且当中断返回时重新取指。显然，从中断返回后的指令到中断前的指令之间，比无中断时有更长的延迟间隔。这样，如果寄存器没有单分配，就可能产生错误结果。下例可说明这一点。

例　没有单独分配寄存器代码：多次使用 A1 寄存器。

```
LDW    .D1    *A0, A1
ADD    .L1    A1, A2,A3
NOP    3
MPY    .M1    A1, A4, A5        ; 利用新的 A1 值
```

假设进入程序之前 A1＝0，A0 指向数值 10 的地址。A1 在执行 LDW(LDW 的延迟间隙为 4，参见 2.2.6 节)的 4 个周期后才被修改为 10，而 ADD 与 LDW 只有一个功能单元等待时间(参见 2.2.6 节)，所以执行 ADD 的结果是 A2＋A1(值为 0)送 A3。然而，如果一个中断

发生在 LDW 与 ADD 之间，当中断结束后返回到 ADD，这时 A1 不再为 0，而是 10，执行 ADD 的结果为 A2＋A1(10)送 A3，显然是不正确的结果。下例中采用单分配方法可解决这一问题。

例 单独分配寄存器代码。

```
LDW     .D1     *A0, A6
ADD     .L1     A1, A2, A3
NOP     3
MPY     .M1     A6, A4, A5      ; uses A6
```

因为 A1 寄存器仅分配一个值，作为 ADD 的一个输入，与 LDW 结果无关，不管是否有中断发生 A1 值都不变，故不会产生错误结果。

4) 中断嵌套

通常当 CPU 进入一个中断服务程序时，其它中断均被禁止。然而，当中断服务程序是可屏蔽中断 INT4～INT15 之一时，NMI 可以中断一个可屏蔽中断的执行过程。换句话说，NMI 可以中断一个可屏蔽中断，但 NMI 或可屏蔽中断均不能中断一个 NMI。

有时希望一个中断服务程序被另一个更高级别的中断所中断，尽管中断服务程序默认不允许被除 NMI 之外的中断所打断，但在软件控制下实现中断嵌套是可能的。这一过程要求如下操作：把原来的 IRP(或者 NRP)和 IER 保存到中断不使用的存储器或寄存器中，通过 ISR 建立一套新的使能位，保存原来的 CSR 后置位 GIE。

5) 手工中断处理

中断响应过程除 CPU 自动检测、自动进入中断服务程序外，还可以手工实现这一过程，即通过手工检测 IFR 和 IER 的状态，然后跳转到 ISTP 指向的地址。下例给出了手工介入的中断处理代码。

例 手工介入的中断处理。

```
        MVC     ISTP，B2            ；获取最高优先级中断的 ISFP 地址
        EXTU    B2，23，27，B1      ；抽出 ISTP 的 HPEINT 位段
[B1]    B       B2                  ；跳转到中断服务程序
[B1]    MVK     1，A0               ；置位 A0 寄存器 bit 0
[B1]    MVK     RET_ADR，B2         ；创建中断程序的返回地址
[B1]    MVKH    RET_ADR，B2         ；
[B1]    MVC     B2，IRP             ；把返回地址保存到 IRP 中
[B1]    SHL     A0，B1，B1          ；创建 ICR 寄存器的值
[B1]    MVC     B1，ICR             ；清除中断标志
RET_ADR：中断返回后继续执行的代码
```

6) 陷阱

陷阱很像中断，所不同的是陷阱由软件建立和控制。陷阱的条件可以存储在 A1、A2、B0、B1 和 B2 的任何条件寄存器内，当陷阱条件为真时，一条跳转指令使 CPU 转入陷阱处理程序，处理结束后返回。下面为陷阱调用和返回代码。

例 陷阱调用代码。

```
[A1] MVK     TRAP_HANDLER，B0        ；创建陷阱地址
[A1] MVKH    TRAP_HANDLER，B0
```

```
[A1] B       B0                 ; 跳转到陷阱处理程序
[A1] MVC     CSR，B0            ; 保存 CSR
[A1] AND     -2，B0，B1         ; 清零 GIE 位，禁止中断
[A1] MVC     B1，CSR            ; 写到 CSR 寄存器中
[A1] MVK     TRAP_RETURN，B1    ; 创建陷阱返回地址
[A1] MVKH    TRAP_RETURN，B1
TRAP_RETURN：陷阱返回后继续执行的程序
```

代码中 A1 为陷阱条件。程序开始时，B0 存放陷阱处理程序首地址，在跳转延迟间隙期间，B0 保存 CSR 内容，以便返回时恢复 CSR。

例　陷阱返回代码。

```
B      B1          ; 返回
MVC    B0，CSR     ; 恢复 CSR
NOP4               ; 延迟间隙
```

2.2.5　片内外设资源

TMS320C6000 中不同芯片的片内外设资源有所不同，表 2.46 中总结了这些芯片的片内外设资源。接下来分别对这些片内外设作较为详细(对于一些较常用外设)或概括性的介绍。

表 2.46　TMS320C6000 的片内外设资源

外　设	C6201	C6202(B) C6203(B)	C6204	C6205	C621x	C6414	C6415	C6701	C671x
DMA	有	有	有	有	无	无	无	有	无
EDMA	无	无	无	无	有	有	有	无	有
主机口(HPI)	有	无	无	无	有	有	有	有	有
扩展总线(XBUS)	无	有	有	无	无	无	无	无	无
PCI	无	无	无	有	无	无	有	无	无
外部存储器接口(EMIF)	1	1	1	1	1	2	2	1	1
BOOT 设置	有	有	有	有	有	有	有	有	有
多通道缓冲串口(McBSP)	2	3	2	2	2	3	3	2	2
UTOPIA	无	无	无	无	无	无	有	无	无
中断选择	有	有	有	有	有	有	有	有	有
32 位定时器	2	2	2	2	2	3	3	2	2
Power - down 逻辑	有	有	有	有	有	有	有	有	有
GPIO	无	无	无	无	无	有	有	无	无

1．直接存储器访问(DMA)

C6000 的 DMA 控制器具有四个可独立编程的 DMA 通道，还有一个辅助通道用来满足主机接口的要求。C6000 的 DMA 控制器具有以下特点：

• 高吞吐率：可以按 CPU 时钟周期进行数据传递。

• 四个通道：DMA 控制器可以同时控制四个独立数据块的传送。

• 辅助通道：此通道允许主机口访问 CPU 的存储器空间，辅助通道相对于其它通道和 CPU 的优先级可以设置。

• 单通道分裂(split - channel)操作：利用单个通道就可以与一个外设同时进行数据的读出和写入，就好像存在两个 DMA 通道一样。

• 多帧(Multiframe)传输：传送的每个数据块可以包含多个帧，每个数据块中帧数和每帧的数据元素数都是可编程的。

• 可编程的优先级：每一 DMA 通道相对于 CPU 的优先级是独立可编程的。

• 可编程的地址产生：每个通道的源地址寄存器和目的地址寄存器对于每次数据传送具有可配置的地址调整值。地址可以保持常量、递增、递减或按预设值进行调整。

• 32 位地址范围：DMA 控制器可以对存储器映射空间中的任一区域进行访问，这些区域包括：片内数据存储区、片内程序存储区(作为映射存储器而不是 cache)、片内外设、通过 EMIF 接口的外部存储器和扩展总线上的扩展存储器等。

• 可编程的传输字宽：每一通道可以独立配置传输字宽，字宽可以选择为字节、半字(16 bit)或字(32 bit)。

• 自动初始化：一个数据块的传送一旦完成后，DMA 通道能够自动初始化其本身以进行下一个数据块传送，即启动链式 DMA。

• 事件同步：每一读、写或帧传送都可以由指定的事件触发。

• 中断产生：当一帧或一个数据块传送完毕，或是出现错误情况时，每一个通道都可以向 CPU 发出中断。

1) DMA 寄存器

DMA 寄存器用来配置 DMA 控制器的操作，所有这些寄存器都映射到存储器空间中。表 2.47 列出了对这些 DMA 寄存器的描述。

表 2.47 DMA 寄存器

缩 写	寄存器名	描 述
AUXCTL	DMA 辅助控制寄存器	配置 DMA 通道和 CPU 的优先级
PRICTLn(n=0～3)	DMA 通道 n 主控寄存器	控制 DMA 通道的操作
SECCTLn(n=0～3)	DMA 通道 n 副控寄存器	控制 DMA 通道的操作
SRC n(n=0～3)	DMA 通道 n 的源地址寄存器	保存下一次读传送的数据源地址
DST n(n=0～3)	DMA 通道 n 的目的地址寄存器	保存下一次写传送的数据目的地址
XFRCNT n(n=0～3)	DMA 通道 n 的传送计数器	配置传送数据块的帧数和每帧的元素数
GBLADDR(A～D)	DMA 全局地址寄存器 A～D	用作 DMA 通道自动初始化的地址重载寄存器或 DMA 分裂(split)通道的地址寄存器
GBLCNT(A～B)	DMA 全局计数重载寄存器 A～B	用作 DMA 通道自动初始化或多帧操作时，传送计数器的重载寄存器
GBLIDX(A～B)	DMA 全局地址修改寄存器 A～B	用作 DMA 通道的可编程地址调整值

下面详细介绍主控寄存器 PRICTL 和副控寄存器 SECCTL 中各位段的定义(参见表 2.48 和表 2.49)，其它寄存器的使用比较简单，本书不再一一详述，对后文碰到的寄存器再作说明。

表 2.48　DMA 通道主控寄存器的位定义

位	名　称	描　　述
31、30	DST RELOAD	通道自动初始化时的目的地址重载设置：00＝不重载，01＝用 GBLADDRB 重载，10＝用 GBLADDRC 重载，11＝用 GBLADDRD 重载
29、28	SRC RELOAD	通道自动初始化时的源地址重载设置：00＝不重载，01＝用 GBLADDRB 重载，10＝用 GBLADDRC 重载，11＝用 GBLADDRD 重载
27	EMOD	仿真模式：0＝在仿真器停止期间 DMA 通道继续运行，1＝在仿真器停止期间 DMA 通道暂停
26	FS	帧同步：0＝禁止，1＝利用 RSYNC 指定的事件来同步整个帧
25	TCINT	传送中断：0＝中断禁止，1＝中断使能
24	PRI	DMA 与 CPU 的优先级模式：0＝CPU 优先，1＝DMA 优先
23～19	WSYNC	写传送同步：00000＝无同步事件，其它＝设置同步事件
18～14	RSYNC	读传送同步：00000＝无同步事件，其它＝设置同步事件
13	INDEX	选择哪个 DMA 全局地址修改寄存器作为可编程地址调整值：0＝选择 GBLIDXA，1＝选择 GBLIDXB
12	CNT RELOAD	自动初始化和多帧传送时，选择传送计数器的重载寄存器：0＝用 GBLCNTA 重载，1＝用 GBLCNTB 重载
11、10	SPLIT	通道分裂模式设置：00＝通道分裂禁止，01＝通道分裂使能(利用 GBLADDRA 作为分裂地址)，10＝通道分裂使能(利用 GBLADDRB 作为分裂地址)，11＝通道分裂使能(利用GBLADDRC作为分裂地址)
9、8	ESIZE	数据元素字宽设置：00＝32 bit，01＝16 bit，10＝8 bit，11＝保留
7、6	DST DIR	每一数据元素传送完成后目的地址的修改模式：00＝不修改，01＝按数据元素字宽递增，10＝按数据元素字宽递减，11＝利用由 INDEX 位指定的 GBLIDX 进行地址调整
5、4	SRC DIR	每一数据元素读传送完成后源地址的修改模式：00＝不修改，01＝按数据元素字宽递增，10＝按数据元素字宽递减，11＝利用由 INDEX 位指定的 GBLIDX 进行地址调整
3、2	STATUS	DMA 通道状态：00＝停止，01＝运行(不带自动初始化)，10＝暂停，11＝运行(带自动初始化)
1、0	START	操作 DMA 通道：00＝停止，01＝启动 DMA(不带自动初始化)，10＝暂停，11＝启动 DMA(带自动初始化)

表 2.49　DMA 通道副控寄存器的位定义

位	名　称	描　　述
31～22	Reversed	保留
21	WSPOL	选择写同步事件(外部中断)的极性：1=低有效，0=高有效 此位只对 C6202(B)/C6203(B)/C6204/C6205 有效
20	RSPOL	选择读和帧同步事件(外部中断)的极性：1=低有效，0=高有效 此位只对 C6202(B)/C6203(B)/C6204/C6205 有效
19	FSIG	电平/边沿触发模式选择：0=边沿触发(FS=1 或 0)，1=电平触发(只有当 FS=1 时) 此位只对 C6202(B)/C6203(B)/C6204/C6205 有效
18～16	DMAC EN	设置 DMAC 管脚的状态
15	WSYNC CLR	清除写同步状态(写 1)
14	WSYNC STAT	写同步状态：0=写同步信号没有检测到，1=写同步信号检测到
13	RSYNC CLR	清除读同步状态(写 1)
12	RSYNC STAT	读同步状态：0=读同步信号没有检测到，1=读同步信号检测到
11	WDROP IE	写拥挤中断使能：0=禁止，1=使能
10	WDROP COND	写拥挤状态：0=WDROP 状态没有检测到，1=WDROP 状态检测到
9	RDROP IE	读拥挤中断使能：0=禁止，1=使能
8	RDROP COND	读拥挤状态：0=RDROP 状态没有检测到，1=RDROP 状态检测到
7	BLOCK IE	数据块传送完成中断使能：0=禁止，1=使能
6	BLOCK COND	数据块传送完成状态：0=BLOCK 状态没有检测到，1=BLOCK 状态检测到
5	LAST IE	最后一帧准备好传送中断使能：0=禁止，1=使能
4	LAST COND	最后一帧准备好传送状态：0=LAST 状态没有检测到，1=LAST 状态检测到
3	FRAME IE	一帧传送完成中断使能：0=禁止，1=使能
2	FRAME COND	一帧传送完成状态：0=FRAME 状态没有检测到，1=FRAME 状态检测到
1	SX IE	分裂传输中发送过分超前接收中断使能：0=禁止，1=使能
0	SX COND	分裂传输状态：0=SX 状态没有检测到，1=SX 状态检测到

2) 设置 DMA 的启动步骤

按以下步骤可完成设置 DMA 的传送启动：

(1) 设置主控寄存器的 START=00；

(2) 设置副控寄存器的 WSYNC CLR=1 和 RSYNC CLR=1；

(3) 设置源/目的地址寄存器和传送计数器；

(4) 设置主控和副控寄存器的其它所需位段；

(5) 向主控寄存器的 START 位段写入：

01=启动不带自动初始化的 DMA 传送(非链式 DMA)；

11=启动带自动初始化的 DMA 传送(链式 DMA)。

对于自动初始化模式，还需要设置重载寄存器：传送计数器重载寄存器和源/目的地址重载寄存器。这些重载寄存器都有两个(或以上)可供选择，由主控寄存器的 DST RELOAD、SRC RELOAD 和 CNT RELOAD 位段来设置。一旦当前的数据块传送结束，该通道的这些寄存器的值就会被重载寄存器的值所替代(或不重载，其值保持不变)。对于传送计数器的 ELEMENTCOUNT 位段(指定帧的元素数)，重载发生在每一帧数据传送完成后，对于多帧传送，不论是否使用自动初始化模式，ELEMENTCOUNT 的重载是必须的。

DMA 通道传送计数器 XFRCNT 包含两个位段，分别对当前传送数据块的帧数和每帧的数据元素数进行计数。XFRCNT 和 GBLCNT 具有相同的位段定义，bit15～bit0 为 ELEMENT COUNT 位段，bit31～bit16 为 FRAME COUNT 位段。

- FRAME COUNT：这个位段内的 16 bit 无符号数对当前传送数据块的帧数进行计数，每个数据块的最大帧数为 65 535。当一帧内的最后一个读操作完成后，此值进行递减。如果自动初始化模式使能，当数据块的最后一帧传送完成后，整个计数器会被 CNT RELOAD 位段指定的 GBLCNT 来重载。FRAME COUNT 的初始值为 0 或 1，其含义一样，都作为单帧数据块。

- ELEMENT COUNT：此位段内的 16 bit 无符号数对当前传送帧的数据元素个数进行计数。每个数据元素的读传送完成后，该值递减。每帧的最大数据元素数目为 65 535。每帧的最后一个数据元素传送完成后，ELEMENT COUNT 会被 CNT RELOAD 位段所指定的 GBLCNT 的低 16 bit 值重载，因此自动初始化和多帧模式都需要对 ELEMENT COUNT 进行重载，即必须设置好主控寄存器的 CNT RELOAD 位段及其指定的全局计数重载寄存器 GBLCNT。如果 ELEMENT COUNT 的初值为零，则操作无效。

DMA 通道的源/目的地址寄存器 SRC/DST 都是 32 bit，分别存放下次读/写操作的地址。DMA 控制器对每次数据读/写传送后，都对源/目的地址重新调整，地址调整可按数据字宽进行递增、递减、保持不变或使用全局地址修改寄存器来进行(由主控寄存器的 SRC DIR 和 DST DIR 位段进行设置)。

主控寄存器的 INDEX 位段选择 GBLIDX。DMA 全局地址修改寄存器(GBLIDX)包含两个位段：ELEMENT INDEX(bit15~bit0)和 FRAME INDEX(bit31～bit16)。全局索引寄存器的 ELEMENT INDEX 位段用来决定源或目的地址在每次数据(除一帧的最后一个数据外)传送完成后的递增或递减的偏移量(字节数)。FRAME INDEX 用来决定一帧中最后一个数据读或写传送完成后的地址调整值(即在当前地址加上此值)。这两个位段都是 16 bit 符号数，范围为-32 768～32 767。

3) DMA 传送同步

DMA 数据传送可以由某一事件来触发同步，这个事件可以为内部外设产生的中断或外部管脚中断等。每一通道可以有三种同步方式：

- 读同步：每次读操作都等待指定的事件发生后才进行；
- 写同步：每次写操作都等待指定的事件发生后才进行；
- 帧同步：每一帧的数据传送都等待指定的事件发生后才进行。

触发事件由主控寄存器的 RSYNC(读同步)和 WSYNC(写同步)位段来指定。如果主控寄

存器的 FS＝1，则 RSYNC 所指定的事件为帧同步事件，此时，WSYNC 必须设置为 00000b。如果 DMA 通道为分裂模式(SPLIT ≠ 00)，RSYNC 和 WSYNC 必须都设置为非零值。表 2.50 中列出了 RSYNC 和 WSYNC 的设置值与同步事件的对应关系。

表 2.50　同步事件设置

事件号(Binary)	事 件 名	事 件 描 述
00000	None	无同步事件
00001	TINT0	定时器 0 中断
00010	TINT1	定时器 1 中断
00011	SD_INT	EMIF SDRAM 定时器中断
00100	EXT_INT4	外中断管脚 4
00101	EXT_INT5	外中断管脚 5
00110	EXT_INT6	外中断管脚 6
00111	EXT_INT7	外中断管脚 7
01000	DMA_INT0	DMA 通道 0 中断
01001	DMA_INT1	DMA 通道 1 中断
01010	DMA_INT2	DMA 通道 2 中断
01011	DMA_INT3	DMA 通道 3 中断
01100	XEVT0	串口 0 发送事件
01101	REVT0	串口 0 接收事件
01110	XEVT1	串口 1 发送事件
01111	REVT1	串口 1 接收事件
10000	DSPINT	主机向 DSP 发出的中断
10001	XEVT2	串口 2 发送事件
10010	REVT2	串口 2 接收事件
其它	保留	

DMA 通道副控寄存器的 RSYNC CLR/WSYNC CLR 和 RSYNC STAT/WSYNC STAT 位用来控制事件同步操作。当指定的事件发生时，RSYNC STAT(读同步)或 WSYNC STAT(写同步)就会置位，从而触发读或写传送操作。当触发的操作完成后，此位自动清零。向 RSYNC CLR 或 WSYNC CLR 写入 1 清除同步事件状态。向 RSYNC STAT 或 WSYNC STAT 写入 1 进行手工设置事件发生，从而强制数据传送开始。向 CLR 和 STAT 位写入 0 无效。C6202(B)/C6203(B)/C6204/C6205 还在副控寄存器中新增加了三个位用来设置同步事件的极性和边沿/电平触发模式。

4) DMA 分裂(Split)通道操作

利用 DMA 分裂通道模式可以只利用一个 DMA 通道就能完成从外部设备中接收数据，并同时向外部设备发送数据的功能。向外部设备的发送和接收地址分别称为分裂目的地址和分裂源地址，由全局地址寄存器来指定。而 DMA 通道的源地址和目的地址与常规 DMA 通道操作的定义一样，保存在 SRC 和 DST 寄存器中。

DMA 分裂通道的操作包括以下两种传送：

• 向分裂目的地址发送数据传送。这是指数据从 DMA 通道的源地址读出(此过程没有

同步事件)，然后把读出的数据写入到分裂目的地址上(此过程必须由 WSYNC 指定同步事件)。每传送一次数据，DMA 通道的源地址会自动调整，而分裂目的地址保持不变。

• 从分裂源地址接收数据传送。这是指数据从分裂源地址读出(此过程必须由 RSYNC 指定同步事件)，然后把读出的数据写入到 DMA 通道的目的地址上(此过程没有同步事件)。每传送一次数据，DMA 通道的目的地址会自动调整，而分裂源地址保持不变。

以上两种传送操作并不要求同步进行，DMA 通道的硬件保证发送数据传送的次数不会超前接收数据传送次数 7 次。

由于每个 DMA 通道只有一个传送计数器，因此收/发数据传送的帧数以及每帧的数据元素数必须相同。为了分裂通道工作正常，RSYNC 和 WSYNC 必须设置为非零值，同时必须禁止帧同步方式。

DMA 通道主控寄存器的 SPLIT 位段指定一个 DMA 全局地址寄存器作为分裂源地址，而分裂目的地址比分裂源地址高一个字(4 个字节地址)。分裂源地址的最低 3 位必须为零，确保分裂源地址在偶数字边界上，而分裂目的地址在奇数字边界上。对于外部设备，用户设计地址译码时必须符合此规定。

5) DMA 中断

DMA 通道可以对 6 种通道状态产生中断，这些通道状态包括：SX(分裂传送中，发送过分超前接收)、FRAME(每一帧传送结束)、LAST(数据块的最后一帧准备传送)、WDROP/RDROP(写/读同步事件拥挤)、BLOCK(数据块的最后一个数据元素传送完成)。这些通道状态是否产生中断，由副控寄存器中对应的中断使能位(IE)来控制。副控寄存器的 SX COND、WDROP COND 和 RDROP COND 为出错告警状态，一旦它们使能且被激活，就会使 DMA 通道进入暂停状态，这些状态直到用户对其写入 0 才会清除。其它通道状态如果对应的使能位(IE)置位则必须手工写 0 才能清除，如果使能位(IE)为 0 则在两个 CPU 时钟周期后会自动清除。

2．扩展的直接存储器访问(EDMA)

EDMA(Enhanced DMA)与 DMA 的功能类似，都是独立于 CPU 完成存储器映射空间之间的数据转移，但其结构与 DMA 控制器有很大差异，主要表现在以下几个方面：

• 扩展到 16 个通道，C64xx 为 64 个通道。

• 2 每一通道的传送都由一个特定的事件来触发。与 DMA 通道不同，EDMA 中每一通道的触发事件是固定的。

• DMA 通道参数都放入寄存器中，而 EDMA 的所有通道传送控制参数都放入 RAM 中，每一事件(对应一通道)都对应一段 RAM 区用来存放此事件的控制参数，一旦某一事件发生，将从 RAM 中读取此事件对应的控制参数，然后将这些参数送往 EDMA 控制器的地址产生器中。

• 类似于 DMA 通道的自动初始化模式，EDMA 控制器提供了一种更为灵活的传送机制，称为“链接”(linking)，它将多个传送参数组链接起来，当完成当前数据块的传送后，会自动装载下一次传送的事件参数。这种链接传送模式可以为某些应用带来方便，例如复杂的排序、循环缓冲等应用。

3．外部存储器接口(EMIF)

C6000 的外部存储器接口 EMIF 可以为多种存储器件提供无缝连接，这些器件包括：同

步突发 SRAM(SBSRAM)；同步 DRAM(SDRAM)；异部器件，包括异步 SRAM、ROM 和 FIFO；外部共享存储器设备。

C64xx 的 EMIF 提供了更加灵活的配置方式，利用一个可编程的同步模式代替 SBSRAM 模式，从而也可对如下同步器件提供无缝接口：ZBT SRAM、同步 FIFO、流水和 flow_thru SBSRAM。

每个 EMIF 存储器映射空间由 4 个 CE0～CE3 空间组成，这 4 个 CE 空间彼此独立，可以进行不同的访问控制。C64xx 提供了两个 EMIF 接口：EMIFA 和 EMIFB。

C6000 的 EMIF 接口特点总结于表 2.51 中。

表 2.51　TMS320C6000 的 EMIF 接口特点

特　点	C6201 C6701	其它 C620x C670x	C621x C671x	C64xx	
				EMIFA	EMIFB
总线宽度/bit	32	32	32	64	64
存储空间块数	4	4	4	4	4
可寻址空间容量(Mbyte)	52	52	512	1024	256
同步时钟	CPU 时钟或(和) 1/2CPU 时钟	1/2CPU 时钟	独立 ECLKIN	独立 ECLKIN，1/4 或 1/6 CPU 时钟	独立 ECLKIN，1/4 或 1/6 CPU 时钟
支持字宽	32 bit; 8/16 bit ROM	32 bit; 8/16 bit ROM	8/16/32 bit	8/16/32 /64 bit	8/16 bit
CE1 空间支持的存储器类型	异步存储器	异步存储器	所有类型	所有类型	所有类型
控制信号	单独的控制信号	同步信号为多工的	所有控制信号都是多工的	所有控制信号都是多工的	所有控制信号都是多工的
支持的同步存储器	SDRAM 和 SBSRAM	SDRAM 或 SBSRAM	SDRAM 和 SBSRAM	所有同步存储器	所有同步存储器
附加的寄存器	—	—	SDEXT	SDEXT CExSEC	SDEXT CExSEC
PDT 支持	无	无	无	有	有
ROM/Flash	√	√	√	√	√
异步存储器 I/O 口	√	√	√	√	√
Pipeline SBSRAM	√	√	√	√	√
Flow thru SBSRAM	—	—	—	√	√
ZBT SRAM	—	—	—	√	√
标准同步 FIFO	—	—	—	√	√
FWFT FIFO	—	—	—	√	√

4．主机口(HPI)

主机口 HPI 是一个并口，外部主机通过此并口可以直接访问 CPU 的存储器映射空间，包括内部存储器和存储器映射的外部设备。主机是此接口的主控方。HPI 与 CPU 存储空间

的连接由 DMA 或 EDMA 来完成，C620x/C670x 中有专门的 DMA 辅助通道完成数据传递，而 C621x/C671x /C64xx 的 HPI 直接连接到片内的地址产生硬件上，没有指定的 EDMA 通道执行 HPI 访问。C64xx 的 HPI 支持 16 bit 或 32 bit 数据宽度，分别称为 HPI16 和 HPI32，它们由复位时的加载和配置模式来选择。C62xx/C67xx 的 HPI 只支持 16 bit 数据宽度。

HPI 利用三个寄存器来完成主机和 C6000 CPU 的通信，这三个寄存器分别是：HPI 控制寄存器(HPIC)、HPI 地址寄存器(HPIA)和 HPI 数据寄存器(HPID)。HPIA 中包含当前主机访问的存储器映射地址，地址为 30 bit 字长，因此最低两位固定为 0。C64xx 的 HPIA 分成两个寄存器：HPI 写地址寄存器(HPIAW)和 HPI 读地址寄存器(HPIAR)，这种读/写地址分开的结构允许主机对不同地址范围分别执行读/写访问。HPID 中包含主机从存储器空间中读取的数据或向存储器空间中写入的数据。HPIC 用来配置主机口的操作，例如设置传输的第一个半字是高 16 bit 还是低 16 bit、中断和 ready 信号等。HPIC 既可以被主机访问也可以被 CPU 访问；HPID 只能被主机访问，而不能被 CPU 访问；HPIA 只能被主机访问(C64xx 的 CPU 也可以访问 HPIA)。

HPI 提供了非常灵活的外部控制信号，充分考虑了不同类型 DSP 的接口需要，因此具有很强的接口能力。

5．扩展总线(XBUS)

扩展总线 XBUS 是主机口 HPI 的替代品，同时又是外部存储器接口 EMIF 的扩展。因此，扩展总线提供的这两种接口——主机口和 I/O 口可以在单系统中共存。扩展总线宽度为 32 bit。

扩展总线的主机口可以提供两种工作模式：异步从属模式和同步主/从模式。在异步从属模式下其操作方式与 C6000 的 HPI 非常相似，只是现在的数据总线宽度变成 32 bit (C64xx 的 HPI 数据总线宽度也是 32 bit)。异步从属模式接口可以与其它利用异步总线的微处理器相连接。同步主/从模式可以方便地与 PCI 桥路芯片以及许多具有同步主/从模式总线的通用微处理器相连接。

扩展总线的 I/O 口也有两种工作模式：异步 I/O 模式和同步 FIFO 模式。XBUS 的 4 个 XCE 空间可以分别选择这些模式，即这两种工作模式在一个系统中可以同时存在。异步 I/O 模式的接口信号时序与 EMIF 类似，也具有高度可编程的特点，在这一模式下，扩展总线接口的四根地址信号使得每个 XCE 空间最多可以挂接 16 个外部设备。同步 FIFO 模式可以提供与一个同步读 FIFO 或四个同步写 FIFO 的无缝连接。利用少量的外部逻辑，每个 XCE 空间可以连接 16 个读 FIFO 和 16 个写 FIFO。

扩展总线的 I/O 口与 DSP 存储空间的连接由 DMA 控制器提供，而扩展总线的主机口与 DSP 存储空间的连接由 DMA 辅助通道来完成。

6．PCI 接口

C6000 的 PCI 接口支持 DSP 与具有 PCI 总线的主机之间的连接。对于 C62xx/C67xx 的 PCI 接口，通过 DMA 辅助通道来完成 PCI 口和 DSP 存储器空间的连接，此 DMA 辅助通道应该具有最高优先级，以使 PCI 接口能完成最大的数据吞吐量。而 C64xx 的 PCI 接口直接连接到 EDMA 的内部地址产生硬件上。PCI 口的这种结构在保证 PCI 主/从数据交换的同时，DMA/EDMA 通道资源仍然可以被其它设备应用。

7．加载配置(Boot Configuration)

C6000 的所有 DSP 芯片都提供了一些外部管脚用来配置芯片在复位时的初始化操作，这些设置包括加载配置(Boot Configuration)、输入时钟模式(倍频)、端口(endian)模式和其它芯片指定的配置。下面只对加载配置(Boot Configuration)进行介绍，其它芯片配置用户可以查阅数据手册。C6000 的 Boot Configuration 配置完成以下复位初始化操作：

(1) 选择存储器映射方式：MAP0 或 MAP1，详见前面的存储器组织介绍；

(2) 选择地址 0 处的外部存储器类型(在 MAP0 模式下)；

(3) 选择对存储器地址 0 处的代码加载方式：

• 无加载：CPU 直接从地址 0 处读取复位中断取指包并开始执行。如果地址 0 处的存储器为 SDRAM，则 CPU 会等待 SDRAM 初始化完成后再去取指、执行。

• ROM 加载：DMA/EDMA 按单帧数据块传送方式把外部 ROM 中的程序复制到 0 地址空间处。程序复制完成后，CPU 退出复位状态，开始从地址 0 处取指令包并执行。对于 C62xx/C67xx，这种加载模式允许用户指定 ROM 的字宽，EMIF 会自动将连续的 8 bit/16 bit 数据合并成 32 bit 指令字进行传送。对于 C620x/C670x，ROM 中的程序要求按 little endian 格式进行存储。而对于 C621x/C671x，ROM 中程序存储的 endian 格式要求与系统使用的格式相同。C64xx 只支持 8 bit ROM 且程序存储的 endian 格式必须与系统使用的格式一致。对于不同的芯片，这一加载过程还有所不同：C620x/C670x 的 DMA 从 CE1 空间中复制 64 K 字节数据到 0 地址空间处；C621x/C671x /C64xx 的 EDMA 从 CE1(C64xx 为 EMIFB CE1)空间的开始处复制 1 K 字节数据到 0 地址空间处。

• 主机加载：CPU 处于复位状态，由外部主机通过主机接口来初始化 CPU 的存储器和配置寄存器，例如 EMIF 的控制寄存器等。当主机完成所有的初始化后，它必须置位 HPIC 寄存器的 DSPINT 位，从而产生中断，唤醒 CPU，CPU 开始从 0 地址空间处取指令包并执行。

所有这些配置都是芯片在复位时根据特定外部管脚的状态来进行的。C6201/C6701 提供了专门的 BOOTMODE[4:0]管脚来设置这些配置。C6202(B)/C6203(B)/C6204 扩展总线 XBUS 上的 XD[4:0]管脚直接映射为 BOOTMODE[4:0]。C6205 EMIF 数据总线上的 ED[4:0]管脚直接映射为 BOOTMODE[4:0]。C621x/C671x 的加载配置只需两位，其主机口上的 HD[4:3]管脚映射为 BOOTMODE[4:3]。C64xx 的加载配置也只需两位，EMIFB 地址总线上的 BEA[19:18]管脚决定加载配置。表 2.52 列出了 C620x/C670x 的加载配置，表 2.53 列出了 C621x/C671x 和 C64xx 的加载配置。

表 2. 52　C620x/C670x 的加载配置

BOOTMODE[4:0]	存储器映射	地址 0 处的存储器	加载方式
00000	MAP0	SDRAM：4 个 8 bit Banks(SDWID=0)	无
00001	MAP0	SDRAM：2 个 16 bit Banks(SDWID=1)	无
00010	MAP0	32 bit 异步设备，默认时序	无
00011	MAP0	1/2×速率 SBSRAM	无
00100	MAP0	1×速率 SBSRAM	无
00101	MAP1	内部存储器	无
00110	MAP0	外部，默认值	主机加载(HPI/XBUS/PCI)

续表

BOOTMODE[4:0]	存储器映射	地址 0 处的存储器	加载方式
00111	MAP1	内部存储器	主机加载(HPI/XBUS/PCI)
01000	MAP0	SDRAM：4 个 8 bit Banks(SDWID=0)	8 bit ROM 加载
01001	MAP0	SDRAM：2 个 16 bit Banks(SDWID=1)	8 bit ROM 加载
01010	MAP0	32 bit 异步设备，默认时序	8 bit ROM 加载
01011	MAP0	1/2×速率 SBSRAM	8 bit ROM 加载
01100	MAP0	1×速率 SBSRAM	8 bit ROM 加载
01101	MAP1	内部存储器	8 bit ROM 加载
01110	保留		
01111	保留		
10000	MAP0	SDRAM：4 个 8 bit Banks(SDWID=0)	16 bit ROM 加载
10001	MAP0	SDRAM：2 个 16 bit Banks(SDWID=1)	16 bit ROM 加载
10010	MAP0	32 bit 异步设备，默认时序	16 bit ROM 加载
10011	MAP0	1/2×速率 SBSRAM	16 bit ROM 加载
10100	MAP0	1×速率 SBSRAM	16 bit ROM 加载
10101	MAP1	内部存储器	16 bit ROM 加载
10110	保留		
10111	保留		
11000	MAP0	SDRAM：4 个 8 bit Banks(SDWID=0)	32 bit ROM 加载
11001	MAP0	SDRAM：2 个 16 bit Banks(SDWID=1)	32 bit ROM 加载
11010	MAP0	32 bit 异步设备	32 bit ROM 加载
11011	MAP0	1/2×速率 SBSRAM	32 bit ROM 加载
11100	MAP0	1×速率 SBSRAM	32 bit ROM 加载
11101	MAP1	内部存储器	32 bit ROM 加载
11110	保留		
11111	保留		

表 2.53　C621x/C671x 和 C64xx 的加载配置

HD[4:3](C6211/C6711) BOOTMODE[1:0](C6712) BEA[19:18](C64xx)	C621x/671x 加载方式	C64xx 加载方式
00	主机加载	无
01	8 bit ROM 加载	主机加载
10	16 bit ROM 加载	EMIFB 8 bit ROM 加载
11	32 bit ROM 加载	保留

8．多通道缓冲串口(McBSP)

C6000 的多通道缓冲串口 McBSP 是在 C2x、C3x、C5x 和 C54x 的标准串口的基础上发展起来的，McBSP 可以提供如下功能和特点：全双工通信；具有收发双缓冲寄存器，允许连续数据流传送；有收发独立的帧信号和时钟信号；可以直接与工业标准的编/解码器、AIC、

串行 A/D 和 D/A 等设备连接；数据传送可以利用外部时钟或片内可编程的时钟；利用 5 个 DMA 通道的自缓冲能力；支持多种协议设备的直接连接，包括 T1/E1 帧协议、MVIP 交换方式和 ST－BUS 兼容设备(包括 MVIP 帧协议、H.100 帧协议和 SCSA 帧协议)、IOM－2 兼容设备、AC97 兼容设备、IIS 兼容设备和 SPI 设备；具有高达 128 个收发通道；多种字宽选择：8/12/16/20/24/32/ bit；μ 律/A 律压扩；8 bit 数据传送可以选择 LSB 或 MSB 先发；帧同步信号和数据时钟信号的极性都可设置；具有高度可编程的内部时钟和帧同步信号产生。

McBSP 与外部设备的连接由两个通道完成：数据通道和控制通道。数据通道包括一个发送串行数据管脚(DX)和一个接收串行数据管脚(DR)；控制通道包括 4 个同步管脚：接收时钟管脚(CLKR)、发送时钟管脚(CLKX)、接收帧同步管脚(FSR)和发送帧同步管脚(FSX)。McBSP 的接收数据端采用三级缓冲，包括接收移位寄存器(RSR)、接收缓冲寄存器(RBR)和数据接收寄存器(DRR)；McBSP 的发送数据端采用二级缓冲，包括数据发送寄存器(DXR)和发送移位寄存器(XSR)。

CPU 或 DMA/EDMA 向 DXR 寄存器写入要发送的数据，DXR 中的数据通过 XSR 移位输出到 DX 管脚。类似地，DR 管脚上接收到的数据先移位进入 RSR 中，然后被复制到 RBR 中，RBR 中的数据再被复制到 DRR 中，最后 CPU 或 DMA/EDMA 将数据从 DRR 中读走。

McBSP 还有 8 个控制寄存器，用于对 McBSP 的操作模式进行设置，这 8 个控制寄存器分成两组：时钟和帧同步信号产生和控制寄存器组，包括串口控制寄存器(SPCR)、接收控制寄存器(RCR)、发送控制寄存器(XCR)、采样率产生器寄存器(SRGR)和管脚控制寄存器(PCR)；多通道选择寄存器组，包括多通道控制寄存器(MCR)、接收通道使能寄存器(RCER)和发送通道使能寄存器(XCER)。

串口控制寄存器 SPCR 中有两位 RRDY 和 XRDY 分别用来指示接收准备好和发送准备好状态。当 RBR 寄存器中的内容复制到 DRR 寄存器中时，RRDY=1 指示 CPU 或 DMA/EDMA 可以从 DRR 中读出此新数据，一旦 DRR 中的新数据被 CPU 或 DMA/EDMA 读走，RRDY 就会自动清零。当 DXR 寄存器中的内容被复制到 XSR 寄存器中时，XRDY=1 指示 CPU 或 DMA/EDMA 可以向 DXR 中加载新的数据，一旦 CPU 或 DMA/EDMA 把新的数据加载到 DXR 中，DRDY 就会自动清零。

根据 RRDY 和 XRDY 的状态，可以有三种方式从 McBSP 读写数据：

• CPU 通过查询 RRDY 和 XRDY 状态的方式决定从 McBSP 接收和向 McBSP 发送数据；

• RRDY 和 XRDY 可以直接驱动 DMA/EDMA 的同步事件 REVT 和 XEVT，从而可利用 REVT 和 XEVT 事件来启动 DMA/EDMA 通道，实现与 McBSP 之间的数据传送；

• RRDY 和 XRDY 可以驱动 McBSP 的接收中断 RINT 和发送中断 XINT(此时 SPCR 寄存器的 RINTM 和 XINTM 位段必须为 00b)，因此可以采用 CPU 中断方式完成对 McBSP 的数据读写。

利用 McBSP 进行数据传递之前还必须首先对 McBSP 进行复位(包括串口的发送通道、接收通道和采样率产生器等)，设置好所有控制寄存器(包括时钟和帧信号极性和主控方、帧长度、数据单元字宽、压扩模式、帧率和时钟率等)。由于篇幅所限，本书不再详细介绍这些内容，用户可参考相关数据手册。

9．定时器(Timer)

C6000 的 32 bit 通用定时器可以用来完成对事件计时、对事件计数、产生脉冲信号、中断 CPU 和产生 DMA 通道的同步事件。

定时器的时钟既可以采用内部时钟，也可以接收外部时钟输入。定时器有两个外部管脚：一个输入管脚 TINP 和一个输出管脚 TOUT，这两个管脚既可以作为定时器时钟的输入和输出管脚，也可以配置为通用 I/O 口。

利用内部时钟，定时器可以启动一个外部 A/D 转换器，或触发 DMA 控制器开始数据传送；利用外部时钟，定时器可以对外部事件进行计数，当记录的事件达到一定次数后中断 CPU。

每一定时器都有三个寄存器用来控制和配置定时器的操作，这三个定时器寄存器分别为：

- 定时器控制寄存器(CTL)：设置定时器的操作模式、监视定时器的状态和设置 TOUT 管脚的功能；
- 定时器周期寄存器(PRD)：包含定时器需要计数的输入时钟周期数；
- 定时器计数寄存器(CNT)：定时器当前的计数值。

定时器计数寄存器 CNT 对定时器的输入时钟源进行加计数，当 CNT 中计数的值达到 PRD 的值时，CNT 会自动置零。

10．中断选择器

C6000 的外围设备最多可以提供高达 32 种中断源，而 CPU 能够响应的可屏蔽中断只有 12 个。因此，中断选择器的目的就是从这 32 种中断源中选择 12 个作为用户系统所需的中断源，而且也允许用户指定外部中断信号的有效极性。

中断选择器提供了三个寄存器来为每个可屏蔽中断(INT4～INT15)选择中断源以及指定外部中断的极性，这三个寄存器分别为：

- 中断源选择寄存器(高)MUXH：为 CPU 中断 INT10～INT15 选择中断源；
- 中断源选择寄存器(低)MUXL：为 CPU 中断 INT4～INT9 选择中断源；
- 外部中断信号极性寄存器 EXTPOL：选择外部中断信号(EXT_INT4～EXT_INT7)的极性。

这三个寄存器的位定义如图 2.10 所示。XIP 选择外中断信号的极性，XIP=0 表示上升沿有效，XIP=1 表示下降沿有效。INTSEL 指定中断源的选择号，表 2.54 中列出了 C6000 的所有中断源。芯片复位后默认的 CPU 中断与中断源的对应关系如表 2.55 所示。



图 2.10　中断选择器的寄存器的位定义

表 2.54　C6000 的所有可屏蔽中断源

中断源选择号 (Binary)	中断信号缩写	描　述
00000	DSPINT	主机发向 DSP 的中断
00001	TINT0	定时器 0 中断
00010	TINT1	定时器 1 中断
00011	SD_INT SD_INTA	EMIF SDRAM 定时器中断 EMIFA SDRAM 定时器中断(C64xx)
00100	EXT_INT4 GPINT4/EXT_INT4	外部中断 4 GPIO 中断 4/外部中断 4(C64xx)
00101	EXT_INT5 GPINT5/EXT_INT5	外部中断 5 GPIO 中断 5/外部中断 5(C64xx)
00110	EXT_INT6 GPINT6/EXT_INT6	外部中断 6 GPIO 中断 6/外部中断 6(C64xx)
00111	EXT_INT7 GPINT7/EXT_INT7	外部中断 7 GPIO 中断 7/外部中断 7(C64xx)
01000	DMA_INT0 EDMA_INT EDMA_INT	DMA 通道 0 中断(C620x/C670x) EDMA 通道(0～15)中断(C621x/C671x) EDMA 通道(0～63)中断(C64xx)
01001	DMA_INT1 Reversed	DMA 通道 1 中断(C620x/C670x) 保留(C621x/C671x/ C64xx)
01010	DMA_INT2 Reversed	DMA 通道 2 中断(C620x/C670x) 保留(C621x/C671x/ C64xx)
01011	DMA_INT3	DMA 通道 3 中断(C620x/C670x) 保留(C621x/C671x/ C64xx)
01100	XINT0	McBSP0 发送中断
01101	RINT0	McBSP0 接收中断
01110	XINT1	McBSP1 发送中断
01111	RINT1	McBSP1 接收中断
10000	Reversed GPINT0	保留(C62xx/C67xx) GPIO 中断 0(C64xx)
10001	XINT2 PCI_WAKEUP	McBSP2 发送中断(C6202(B)/C6203(B)/C64xx) PCI 唤醒中断(C6205)
10010	RINT2 ADMA_HLT	McBSP2 接收中断(C6202(B)/C6203(B)/C64xx) 辅助 DMA 停止中断(C6205)
10011	TINT2	定时器 2 中断(C64xx)
10100	SD_INTB	EMIFB SDRAM 定时器中断(C64xx)
10101	PCI_WAKEUP	PCI 唤醒中断(C64xx)
10111	UINT	UTOPIA 中断(C64xx)
其它	保留	

表 2.55　复位后默认的中断与中断源的对应关系

CPU 中断	中断信号缩写	描　述
INT4	EXT_INT4	外部中断 4
INT5	EXT_INT5	外部中断 5
INT6	EXT_INT6	外部中断 6
INT7	EXT_INT7	外部中断 7
INT8	DMA_INT0 EDMA_INT	DMA 通道 0 中断 EDMA 中断
INT9	DMA_INT1	DMA 通道 1 中断(C620x/C670x)
INT10	SD_INT SD_INTA	EMIF SDRAM 定时器中断(C62xx/C67xx) EMIFA SDRAM 定时器中断(C64xx)
INT11	DMA_INT2	DMA 通道 2 中断(C620x/C670x)
INT12	DMA_INT3	DMA 通道 3 中断(C620x/C670x)
INT13	DSPINT	主机向 DSP 发出的中断
INT14	TINT0	定时器 0 中断
INT15	TINT1	定时器 1 中断

用户应该在芯片复位后、中断使能前配置中断源选择寄存器。用户在配置完中断源选择寄存器后，等待一段延迟后清除整个中断标志寄存器，以免受到配置过程中产生的“虚假中断条件”的影响。

11．低功耗控制逻辑(Power－down)

CMOS 器件的大部分功耗都来自于逻辑电路的开关操作。C6000 提供了三种功耗控制模式：PD1、PD2 和 PD3。在不会引起数据丢失的前提下，关闭掉芯片中部分或所有逻辑电路的开关操作，可以大大降低芯片的功耗。这三种模式的功能如下：

• PD1 模式：关闭掉输入 CPU 的时钟，CPU 停止(除中断逻辑)，但 DMA/EDMA 仍然可以继续进行外设和内部存储器之间的数据传递。在此模式下，芯片可以由使能的中断或任何中断(使能或非使能)唤醒。

• PD2 模式：PLL 的输出时钟被停止，导致整个芯片停止，只能由 Reset 唤醒芯片。

• PD3 模式：类似 PD2，但 PD3 模式还断开 PLL 的输入时钟源(CLKIN)，导致整个芯片停止，只能由 Reset 唤醒芯片。

C620x/C670x 在 PD2 和 PD3 模式下还驱动外部管脚 PD 为高，用来通知外部设备当前芯片的功耗模式。C621x/C671x/C64xx 没有此 PD 管脚。

这三种低功耗模式的选择由控制和状态寄存器 CSR 的 PRWD 位段指定，表 2.56 中列出了 PRWD 位段指定的功耗模式。

表 2.56　PRWD 位段指定的功耗模式

PRWD(Binary)	Power－down 模式	唤　醒　方　式
000000	无 Power-down	—
001001	PD1	由使能的中断唤醒
010001	PD1	由使能的或非使能的中断唤醒
011010	PD2	由芯片复位唤醒
011100	PD3	由芯片复位唤醒
其它	保留	

C6202(B)/C6203(B)还允许用户关闭掉系统中不需要的片内外设。这些片内外设的关闭由外设低功耗控制寄存器(PDCTL)的相应位控制(置位)。表 2.57 中列出了 PDCTL 的控制位。

表 2.57　PDCTL 的外设关闭控制位

位	名　称	描　　述
4	PDMCSP2	使能/禁止片内 McBSP2 时钟，0=使能，1=禁止
3	PDMCSP1	使能/禁止片内 McBSP1 时钟，0=使能，1=禁止
2	PDMCSP0	使能/禁止片内 McBSP0 时钟，0=使能，1=禁止
1	PDEMIF	使能/禁止片内 EMIF 时钟，0=使能，1=禁止
0	PDDMA	使能/禁止片内 DMA 时钟，0=使能，1=禁止

除了利用低功耗模式使芯片进入低功耗状态外，利用 IDLE 指令也可达到这一目的。IDLE 指令只能由中断服务来终止。

12．ATM 通用测试和操作接口(UTOPIA)

C6000 的 UTOPIA 外设是 ATM 控制器的被控端，支持 UTOPIA 2 级接口，允许数据以 8 bit、50 MHz 的吞吐量进行发送和接收。UTOPIA 接口依靠 ATM 主控端提供所需的控制信号，例如时钟信号、使能和地址等，仅支持数据包级握手。

UTOPIA 接口包括一个发送接口和一个接收接口。发送接口中包含一个数据发送队列(UXQ)，接收接口中包含一个数据接收队列(URQ)，这些数据队列既可以被 CPU 访问，也可以被 EDMA 访问(推荐)。通过 UTOPIA 的数据传送以 ATM 数据包的方式进行，标准的 ATM 数据包为 53 个字节(5 字节头信息+48 字节有效数据)，UTOPIA 还支持非标准的 ATM 数据包(54 字节～64 字节)。关于 ATM 数据包的详细定义，用户可查阅相关的 ATM 标准。

UTOPIA 具有以下配置寄存器：UTOPIA 控制寄存器(UCR)、UTOPIA 中断使能寄存器(UIER)、UTOPIA 中断挂起寄存器(UIPR)、时钟检测寄存器(CDR)、出错中断使能寄存器(EIER)和出错中断挂起寄存器(EIPR)，用于配置 UTOPIA 接口的各种操作。

EDMA 和 CPU 都可服务于 UTOPIA 接口，用于向 UTOPIA 提供发送数据和从 UTOPIA 接收数据。UTOPIA 能向 EDMA 通道提供两个同步事件——UXEVT(发送)和 UREVT(接收)。UXEVT 应用于 EDMA 通道 40，当数据发送队列中至少空一个数据包的空间时就释放此事件；UREVT 应用于 EDMA 通道 32，当数据接收队列中有一完整的数据包到达时就释放此事件。UTOPIA 还能向 CPU 发出中断 UINT，发送和接收端都可以利用此中断，由 UTOPIA 的中断使能寄存器(UIER)的相应位使能或禁止此发送和接收中断。

EDMA(推荐)用于 UTOPIA 接口的数据发送和接收时，需要配置如下参数：

- ESIZE=00b：32 bit 数据元素；
- SUM=01b：源地址自动递增；
- DUM=01b：目的地址自动递增；
- FS=1：帧同步传送，每一次 UXEVT(或 UREVT)同步事件的发生，都向数据发送对列 UXQ 发送(或从数据接收对列 URQ 读走)一个完整的 ATM 数据包；
- SRC 地址=源缓冲区的开始地址(发送)；

  SRC 地址=URQ 的开始地址(接收)；
- DST 地址=UXQ 的开始地址(发送)；

DST 地址=目的缓冲区的开始地址(接收)；

- Element Count=14，15 或 16：传送整个数据包；
- 其它 EDMA 参数。

CPU 也可用于 UTOPIA 接口的数据发送或接收。当 UTOPIA 中断使能且发生时，CPU 利用中断服务程序完成如下操作：

(1) 读 UIPR 寄存器的值，判断是哪个数据队列(URQ 或 UXQ)产生的中断；

(2) 从 URQ 中读走一个完整的 ATM 数据包(URQ 产生的中断)，或向 UXQ 中写入一个完整的 ATM 数据包(UXQ 产生的中断)；

(3) 清除 UIPR 的相应中断标志位。写入 1 清除中断标志，写入 0 无效。

13．通用 I/O(GPIO)外设

C64xx 的 GPIO 外设提供了一套管脚(部分管脚与其它设备是复用的)可以配置成通用输入或输出管脚，这些管脚共有 16 个：GP0～GP15。当配置为输出管脚时，用户通过向 GPIO 值寄存器(GPVAL)的对应位写入 1 或 0 来驱动此管脚的输出状态；当配置为输入管脚时，用户通过读 GPVAL 中对应位的值来检测此管脚的输入状态。输入管脚的状态还可以用来产生 CPU 中断或 EDMA 同步事件。所有的 GPx 都可作为 EDMA 的同步事件产生源，而只有 GP0 和 GP[4:7]可作为 CPU 的中断源(由中断选择器来选择)。表 2.58 中列出了 GPIO 寄存器，用户通过这些寄存器来配置 GPIO 外设。

表 2.58　GPIO 寄存器

寄存器缩写	寄存器名称	描　述
GPEN	GPIO 使能寄存器	bit0～bit15 对应 GP0～GP15 管脚，分别使能/禁止通用 I/O 管脚，1＝使能，0＝禁止
GPDIR	GPIO 方向寄存器	bit0～bit15 分别指定 GP0～GP15 作为通过 I/O 管脚的方向，1＝输出，0＝输入
GPVAL	GPIO 值寄存器	bit0～bit15 分别对应 GP0～GP15 管脚，用来驱动输出管脚的状态或指示输入管脚的状态值
GPDH	GPIO 低高转变寄存器	bit0～bit15 分别指示 GP0～GP15 输入管脚是否经历一个由低到高的转变过程，1＝有，0＝无
GPDL	GPIO 高低转变寄存器	bit0～bit15 分别指示 GP0～GP15 输入管脚是否经历一个由高到低的转变过程，1＝有，0＝无
GPHM	GPIO 高屏蔽寄存器	bit0～bit15 分别用来使能/禁止由 GPDH 或 GPVAL 中相应位产生的中断或 EDMA 事件，1＝使能，0＝禁止
GPLM	GPIO 低屏蔽寄存器	bit0～bit15 分别用来使能/禁止由 GPDL 相应位或 GPVAL 相应位的取反值产生的中断或 EDMA 事件，1＝使能，0＝禁止
GPGC	GPIO 全局控制寄存器	配置 GPIO 外设的中断/事件产生模式
GPPOL	GPIO 中断极性寄存器	选择 GP0、GP[4:7]中断/事件源信号的极性，1＝GPx 时信号的下降沿有效，0＝GPx 时信号的上升沿有效

GPIO 外设产生 CPU 中断或 EDMA 同步事件的基本模式有两种：单个源分别产生或按某一逻辑关系产生。GPIO 全局控制寄存器(GPGC)设置 CPU 中断/EDMA 同步事件的各种产生模式。

2.2.6　TMS320C6000 DSP 的汇编指令

TMS320C62xx、TMS320C64xx 和 TMS320C67xx 共享一套指令集。C62xx 的所有指令均对 C67xx 和 C64xx 有效。由于 C67xx 为浮点芯片，因此 C67xx 有自己的一些特定指令，这些特定指令(包括 32 bit 整型乘法、双字存储器读和浮点操作(浮点加、减和乘))在定点 C62xx 芯片上不能执行。同样，C64xx 也具有一些不同于 C62xx 的特定指令，包括双 16 位、四 8 位等操作。下面对 TMS320C6000 的汇编程序编写和混合程序编写(在 C/C++程序中调用汇编函数)作一介绍。

1. 指令形式

C6000 汇编代码的基本形式为

标号：并行符号　[条件]　指令　功能单元　　操作数　；注释

1) 标号

标号用来指示一行代码或一个变量，代表此行代码或变量的存储地址。标号是可选项，标号的第 1 个字符必须在文件的第 1 列。

2) 并行符号

如果一条指令与前面的指令并行执行，这条指令前用并行符号“||”表示。如果一条指令前没有并行符号，表示此指令不与前面的指令并行执行。

3) 条件

所有 C6000 指令都是条件指令。C6000 中有 5 个寄存器可作为条件寄存器：A1、A2、B0、B1 和 B2。方括号内的条件为上述寄存器中的任一寄存器。当条件为真时此行指令才执行；当条件为假时此行指令不执行；如果指令前没有指出条件，此指令总执行。

例　[A1]　当 A1 的值不为 0 时，指令才被执行。

[!A1] 当 A1 的值为 0 时(!A1 为真)，指令才被执行

4) 指令

汇编代码的指令包括伪指令和命令助记符。

伪指令是汇编器命令用来控制汇编过程和定义数据结构等，所有伪指令都以圆点(.)开头。表 2.59 中列出了 C6000 的部分伪指令。

表 2.59　C6000 编译器的部分伪指令

伪　指　令	说　　明
.sect "*name*"	创建数据或代码段
.data	把接下来分配的数据空间放入到.data 段(初始化数据段)
.text	把接下来的代码或数据空间放入到.text 段(可执行代码段)
.double *value1* [, ... , *valuen*]	在当前存储段中初始化一个或多个 64 位 IEEE 双精度浮点数据
.float *value1* [, ... , *valuen*]	在当前存储段中初始化一个或多个 32 位 IEEE 单精度浮点数据
.int　*value1* [, ... , *valuen*] .long　*value1* [, ... , *valuen*] .word　*value1* [, ... , *valuen*]	在当前存储段中初始化一个或多个 32 位整型数据
.short　*value1* [, ... , *valuen*] .half　*value1* [, ... , *valuen*]	在当前存储段中初始化一个或多个 16 位整型数据

续表

伪　指　令	说　　明
.byte　*value1* [*,* ... *,* *valuen*] .char　*value1* [*,* ... *,* *valuen*]	在当前存储段中初始化一个或多个连续的 8 位数据
.bss *symbol,size in bytes*	在.bss 数据段(非初始化数据段)中保留连续的 *size* 个字节空间
symbol .set　*value* *symbol* .equ　*value*	定义一个常数符号 *symbol* 来代表 *value*，对于程序中遇到的所有此符号，汇编器自动用 *value* 来代替
.align *size*	把 SPC 定位到 *size* 字节的边界上
.def *symbol1* [, ... , *symboln*]	声明当前模块中定义的符号能被其它模块使用
.global *symbol1* [, ... ,*symboln*]	声明全局符号
.ref *symbol1* [, ... , *symboln*]	声明在当前模块中用到的在其它模块中定义的符号
.macro　*code*　.endm	定义宏指令

命令助记符是真正的处理器命令，它执行实际的程序操作。表 2.60 中列出了 C62xx/C67xx/C64xx 所有共享的定点命令助记符及其指令形式和执行操作。表 2.61 列出了 C67xx 的特定命令助记符及其指令形式和执行操作。关于 C64xx 的特定命令助记符，请查阅用户手册。表中各符号的含义见表后解释。

表 2.60　C62xx/C67xx/C64xx 的定点命令助记符及其指令形式和执行操作

助记符指令形式		执　行　操　作
算术操作指令		
ABS	src2, dst	abs(src2)→dst，取 src 的绝对值放入 dst 中
ADD[U]	src1, src2, dst	src1 + src2→dst，带后缀 U，表示无符号数相加
SUB[U]	src1, src2, dst	src1– src2→dst，带后缀 U，表示无符号数相减
ADDAB ADDAH ADDAW	src2, src1, dst	src1 分别按字节(×1，B)、半字(×2，H)或字(×4，W)寻址模式与 src2 相加，结果放入 dst 中。如果 src2 为 A4～A7 或 B4～B7，src1 还可按循环寻址模式(AMR 指定)与 src2 相加
SUBAB SUBAH SUBAW	src2, src1, dst	src1 分别按字节(×1，B)、半字(×2，H)或字(×4，W)寻址模式与 src2 相减，结果放入 dst 中。如果 src2 为 A4～A7 或 B4～B7，src1 还可按循环寻址模式(AMR 指定)与 src2 相减
ADDK	cst16, dst	cst16+dst→dst，16 位符号常数 cst16 加到无符号数寄存器 dst 中
ADD2	src1, src2, dst	((lsb16(src1)+lsb16(src2))&FFFFh)|((msb16(src1)+msb16(src2))<<16)→dst，两源操作数的高 16 位和低 16 位分别对应相加，低位的进位不影响高位相加结果，32 bit 结果放入 dst 中
SUB2	src1, src2, dst	((lsb16(src1)- lsb16(src2))&FFFFh)|((msb16(src1)- msb16(src2))<<16)→dst，两源操作数的高 16 位和低 16 位分别对应相减，低位的借位不影响高位相减结果，32 bit 结果放入 dst 中
SUBC	src1, src2, dst	if (src1–src2≥0)　((src1– src2)<< 1)+1→dst　else src1<<1→dst
NORM	src2, dst	把 src2 中的冗余符号位数(不含符号位)放入 dst 中
NEG	src2, dst	0–src2→dst，src2 的负值放入 dst 中
CMPEQ	src1, src2, dst	if (src1== src2)　1→dst　else　0→dst 比较 src1 和 src2，若相等则 dst 置 1，否则置 0
CMPGT[U]	src1, src2, dst	if (src1>src2) 1→dst else 0→dst，后缀 U 表示无符号数比较
CMPLT[U]	src1, src2, dst	if (src1<src2) 1→dst else 0→dst，后缀 U 表示无符号数比较

续表(一)

助记符指令形式		执 行 操 作
MPY[U/US/SU]	src1, src2, dst	lsb16(src1)×lsb16(src2)→dst，src1 和 src2 的低 16 位相乘，结果放入 dst 中，后缀 U 表示两个无符号数相乘，US 表示 src1 是无符号数而 src2 是有符号数，SU 反之，如果省略后缀则表示两个符号数相乘，如果符号数相乘，dst 中的结果会符号扩展
MPYH [U/US/SU]	src1, src2, dst	msb16(src1)×msb16(src2)→dst，src1 和 src2 的高 16 位相乘，结果放在 dst 低位段，其它同上
MPYHL[U] MPYHULS MPYHSLU	src1, src2, dst	msb16(src1)×lsb16(src2)→dst，src1 的高 16 位和 src2 的低 16 位相乘，其它同上
MPYLH[U] MPYLUHS MPYLSHU	src1, src2, dst	lsb16(src1)×msb 16(src2)→dst，src1 的低 16 位和 src2 的高 16 位相乘，其它同上
SMPY	src1, src2, dst	src1 的低 16 bit 与 src2 的低 16 bit 作为两个符号数相乘，结果左移一位，放入 dst 中，若左移结果为 8000 0000h，则把 7FFF FFFFh 放入 dst 中，同时置位 CSR 的 SAT 位
SMPYLH	src1, src2, dst	src1 的低 16 bit 与 src2 的高 16 bit 作为两个符号数相乘，其它同上
SMPYHL	src1, src2, dst	src1 的高 16 bit 与 src2 的低 16 bit 作为两个符号数相乘，其它同上
SMPYH	src1, src2, dst	src1 的高 16 bit 与 src2 的高 16 bit 作为两个符号数相乘，其它同上
SADD	src1, src2, dst	src1 与 src2 求和，结果置入 dst，若结果溢出，则采取饱和值，并置位 CSR 的 SAT 位
SSUB	src1, src2, dst	src1 减去 src2，结果放入 dst 中，其它同 SADD
SAT	src2, dst	将 40 bit 数 src2 转换为 32 bit 数放入 dst 中，若 src2 超出 32 bit 数的表示范围，则取饱和值，并置位 CSR 的 SAT 位
逻辑操作指令		
AND	src1, src2, dst	src1&src2→dst，按位与，结果放入 dst 中
NOT	src2, dst	−1 xor src2→dst，src2 按位取反后放入 dst 中
OR	src1, src2, dst	src1 与 src2 按位或，结果放入 dst 中
XOR	src1, src2, dst	src1 与 src2 按位异或，结果放入 dst
CLR	src2, csta, cstb, dst	src2 从 csta 到 cstb 之间的位段清零后，放入到 dst 中，src2 保持不变，0≤csta, cstb≤31
	src2, src1, dst	src1 的 bit9～5=csta, bit4～0=cstb，执行操作同上
SET	src2, csta, cstb, dst src2, src1, dst	src2 指定位段置 1 并放入 dst 中，方法同 CLR
EXT	src2,csta,cstb,dst	把 src2 的(cstb – csta)～(31– csta)之间的位段抽出来，放入到 dst 的低位段，dst 的高位段符号扩展，0≤csta, cstb≤31
	src2, src1, dst	src1 的 bit9～5=csta, bit4～0=cstb，执行操作同上

续表(二)

助记符指令形式		执　行　操　作
EXTU	src2,csta,cstb,dst	执行操作类似 EXT，但 dst 的高位段填 0
	src2, src1, dst	执行操作类似 EXT，但 dst 的高位段填 0
SHL	src2, src1, dst	src2 左移，移位量由 src1 寄存器的低 6 位或 5 bit 无符号立即数决定，结果放入 dst 中
SSHL	src2, src1, dst	src2 左移，移位量由 src1 寄存器的低 5 位或 5 bit 无符号立即数决定，结果放入 dst，若发生溢出，正数取 7FFF FFFFh，负数取 80000000h
SHR	src2, src1, dst	src2 算术右移，高位符号扩展，移位量由 src1 寄存器的低 6 位或 5 bit 无符号立即数决定，结果置入 dst
SHRU	src2, src1, dst	src2 逻辑右移，高位填零，其它同上
LMBD	src1, src2, dst	搜索 src2 中左起第一个 0 或 1 所在位数(以最左边为 0 计数)，搜索结果放入 dst 中，由 src1 的 bit0 位决定搜索 1 还是 0
程序控制指令		
B	label	label→PFC，程序取指计数器(PFC)从 label 地址处取指
	src2	src2→PFC
	IRP	IRP→PFC, PGIE→GIE，从可屏蔽中断返回
	NRP	NRP→PFC, 1→NMIE, 从不可屏蔽中断返回
NOP	[count]	count 个时钟周期的空操作，1≤count≤9，count 省略，默认为 1
IDLE		芯片进入低功耗等中断状态
加载存储指令		
LDB[U]	间接寻址, dst	把存储器中的 8 bit 数据加载到通用寄存器 dst 的低 8 bit，带后缀 U 表示 dst 的高 24 bit 填 0，否则进行符号扩展
LDH[U]	间接寻址, dst	把存储器中的 16 bit 数据加载到通用寄存器 dst 的低 16 bit，带后缀 U 表示 dst 的高 16 bit 填 0，否则进行符号扩展
LDW	间接寻址, dst	把存储器中的 32 bit 数据加载到通用寄存器 dst 中
STB STH STW	src，间接寻址	把通用寄存器的数据放到存储器中，其它同 LDB/LDH/LDW 指令
MV	src2, dst	0+src2→dst，相当于把 src2 中的数据传给 dst
MVC	src2, dst	src2→dst，实现控制寄存器与通用寄存器之间的数据传递
MVK	cst16, dst	16 bit 常数符号扩展后放入 dst 中
MVKLH	cst32, dst	32 bit 常数的低 16 bit 放入 dst 的高 16 位中，dst 的低 16 位不变
MVKH	cst32, dst	32 bit 常数的高 16 bit 放入 dst 的高 16 位中，dst 的低 16 位不变
MVLH	cst32, dst	32 bit 常数的低 16 bit 符号扩展后放入 dst 中，利用 MVKL 和 MVKH 可以加载一个标号地址。 例　MVKL *label*，A4 　　MVKH *label*，A4
ZERO	dst	0→dst

表 2.61 C67xx 的特定命令助记符及其指令形式和执行操作

指 令 格 式		执 行 操 作
ABSDP	src2, dst	64 位双精度浮点数 src2 取绝对值，放入 dst 中
ABSSP	src2, dst	32 位单精度浮点数 src2 取绝对值，放入 dst 中
ADDAD	src2, src1, dst	src1 按双字寻址模式(×8)与 src2 相加，结果放入 dst 中。如果 src2 为 A4～A7 或 B4～B7，src1 还可按循环寻址模式(AMR 指定)与 src2 相加
ADDDP	src1, src2, dst	src1+src2→dst，64 位双精度浮点数加法
ADDSP	src1, src2, dst	src1+src2→dst，32 位单精度浮点数加法
CMPEQDP	src1, src2, dst	if (src1==src2) 1→dst else 0→dst 64 位双精度浮点数比较
CMPEQSP	src1, src2, dst	if (src1==src2) 1→dst else 0→dst 32 位单精度浮点数比较
CMPGTDP	src1, src2, dst	if (src1>src2) 1→dst else 0→dst 64 位双精度浮点数比较
CMPGTSP	src1, src2, dst	if (src1>src2) 1→dst else 0→dst 32 位单精度浮点数比较
CMPLTDP	src1, src2, dst	if (src1<src2) 1→dst else 0→dst 64 位双精度浮点数比较
CMPLTSP	src1, src2, dst	if (src1<src2) 1→dst else 0→dst 32 位单精度浮点数比较
DPINT	src2, dst	int(src2)→dst，64 位双精度浮点数转换为 32 位整型数，舍入模式由 FADCR 寄存器的 Rmode 位段控制
DPSP	src2, dst	把 64 位双精度浮点数转换为 32 位单精度浮点数
DPTRUNC	src2, dst	把 64 位双精度浮点数转换为 32 位整型数，总是向零方向舍入
INTDP[U]	src2, dst	把 32 位整型数转换为 64 位双精度浮点数，后缀 U 表示无符号数
INTSP[U]	src2, dst	把 32 位整型数转换为 32 位单精度浮点数，后缀 U 表示无符号数
LDDW	间接寻址, dst	把 64 位双字数据从存储器加载到寄存器中
MPYDP	src1, src2, dst	把 src1×src2→dst，64 位双精度浮点数乘法，64 位结果放入 dst 中
MPYI	src1, src2, dst	把 lsb32(src1× src2)→dst，32 位整型数乘法，只取乘积结果的低 32 位放入 dst 中
MPYID	src1, src2, dst	把 32 位整型数乘法，64 位结果放入 dst 中
MPYSP	src1, src2, dst	32 位单精度浮点数乘法，32 位结果放入 dst 中
RCPDP	src2, dst	计算 64 位双精度浮点数倒数的近似值，结果放入 dst 中
RCPSP	src2, dst	计算 32 位单精度浮点数倒数的近似值，结果放入 dst 中
RSQRDP	src2, dst	计算 64 位双精度浮点数平方根倒数的近似值，结果放入 dst 中
RSQRSP	src2, dst	计算 32 位单精度浮点数平方根倒数的近似值，结果放入 dst 中
SPDP	src2, dst	把 32 位单精度浮点数转换为 64 位双精度浮点数
SPINT	src2, dst	把 32 位单精度浮点数转换为 32 位整型数，舍入模式由 FADCR 寄存器的 Rmode 位段控制
SPTRUNC	src2, dst	把 32 位单精度浮点数转换为 32 位整型数，总是向零舍入
SUBDP	src1, src2, dst	src1−src2→dst，64 位双精度浮点数减法
SUBSP	src1, src2, dst	32 位单精度浮点数减法

注：表 2.60 和表 2.61 中各符号的含义：

cstn, csta, cstb　　n bit 常数，常数 a，常数 b

lsb16(x), msb16(x)　　x 的低 16 位，高 16 位

<<, >>　　左移，右移

src1, src2　　源操作数 1，源操作数 2，src2 一定为通用寄存器，而 src1 有时还可以为常数(立即数)，用户可以通过 CCS 的帮助来得到每一个指定的操作数要求，本书不再一一详述

dst　　目的操作数，为通用寄存器

&　　按位与

|　　按位或

5) 功能单元

C6000 有 8 个功能单元：.S1,.S2,.L1,.L2,.M1,.M2,.D1,.D2。不同类型的功能单元完成不同的任务。功能单元都以点(.)开始，后跟一个功能单元类型符。功能单元用来说明指令所使用的资源，也是可选项。编程时可以指定具体使用的功能单元(例如.D1)，也可以只指出功能单元类型(例如.M)，由汇编器安排特定功能单元(例如.M1)，甚至可以不指出功能单元，由汇编器根据助记符自动安排功能单元。

6) 操作数

在汇编代码中，指令对操作数有如下要求：所有指令都需要一个目的操作数，目的操作数一定放在最后；多数指令需要一个或两个源操作数；目的操作数必须与一个源操作数在同一寄存器组；两源操作数可以在同一寄存器组，也可在不同寄存器组。

当某一操作数来自另一寄存器组时，该指令所使用的功能单元后应加一“X”符号，表示此指令使用一条交叉通路。

例　ADD　.L1X　A0，　B1，　A3

C6000 指令使用三种类型操作数来访问数据：寄存器操作数，即寄存器中包含的操作数据；常数操作数；指针操作数，它包含数据的存储器地址。仅存储器访问指令可利用指针操作数，以实现存储器和寄存器之间的数据传递。

7) 注释

使用注释对代码进行说明，注释前可以使用分号(；)，也可以使用星号(＊)，但使用星号时注释必须从第 1 列开始。

2．寻址模式

C6000 指令的寻址模式比较单一，只有间接寻址模式，即以通用寄存器作为基址，而偏移地址可以为通用寄存器或常数。表 2.62 列出了间接寻址的各种表示方法及其功能。

表 2. 62　间接寻址的表示方法及其功能

表示方法	功能说明
*R	以 R 为地址进行寻址，寻址前后 R 的内容保持不变，如果为.D1 功能单元，则 R 为 A0～A15，如果为.D2 功能单元，则 R 为 B0～B15(下同)
*+R[offsetR]/ *- R[offsetR]	以 R±n×offsetR 为地址进行寻址，寻址前后 R 寄存器的内容保持不变，offsetR 也为通用寄存器，而且 offsetR、R 和功能单元.D 必须在同一数据通路。对于字节、半字、字和双字寻址，n 分别为 1、2、4 和 8(下同)
*++R[offsetR]/ *- - R[offsetR]	以 R±n×offsetR 为地址进行寻址，寻址后 R 寄存器的内容被修改为(R±n×offsetR)，其它同上
*R++[offsetR]/*R- - [offsetR]	以 R 为地址进行寻址，寻址后 R 寄存器的内容被修改为 R±n×offsetR，其它同上
*+R[ucst5]/ *- R[ucst5]	以 R±n×ucst5 为地址进行寻址，寻址前后 R 寄存器的内容保持不变。ucst5 为 5 bit 无符号常数。ucst5 也可以放入括号内，此时无论对于字节、半字、字还是双字寻址，n 总为 1(即字节)
*++R[ucst5]/ *- - R[ucst5]	以 R±n×ucst5 为地址进行寻址，寻址后 R 寄存器的内容被修改为 R±n×ucst5。如果偏移量省略，则默认为 1(下同)，其它同上
*R++[ucst5]/ *R- - [ucst5]	以 R 为地址进行寻址，寻址后 R 寄存器的内容被修改为 R±n×ucst5，其它同上
*+B14/B15[ucst15]	以 B14/B15＋n×ucst15 为地址进行寻址，寻址前后 B14/B15 寄存器的内容保持不变。ucst15 为 15 bit 无符号常数

寄存器与存储器之间的数据交换，只能利用如下指令：LDB[U]/LDH[U]/LDW、LDDW(C67xx 指令)用于从存储器向通用寄存器加载数据；STB/STH/STW 用于把寄存器中的数据写到存储器中。C6000 还提供了 ADDAB/ADDAH/ADDAW、SUBAB/SUBAH/SUBAW、ADDAD(C67xx 指令)指令，用来辅助计算存储器地址。

在上述存储器地址计算时，可以采取两种方式：一种是线性寻址，即偏移量经过×1、×2、×4、×8(分别对应字节、半字、字和双字寻址)处理后直接加到基址上；另一种是循环寻址，即偏移量在经过×1、×2、×4、×8(分别对应字节、半字、字和双字寻址)处理后，还要以循环缓冲区的长度取模数后加到基址上。由 AMR 寄存器选择是线性寻址还是循环寻址，并指定循环缓冲区的长度。需要特别说明的是，AMR 寄存器中指定的循环缓冲区的长度是以字节为单位的。所有通用寄存器都可用于线性寻址，而只有当 A4～A7 或 B4～B7 作为基址时才可选择为循环寻址。

例　LDW .D1 *++A4[9], A1

假定此指令执行前相关寄存器的值为：AMR= 0004 0001h，A4=0000 0100h。AMR 寄存器设置 A4 为循环寻址模式，并且循环缓冲区的长度为 2^N=32 字节。由于 LDW 指令为按字寻址，地址偏移量为 9×4=36 字节，又由于是循环寻址且循环缓冲区长度为 32 字节，因此最终地址调整量为 36- 32=4 字节，即此指令执行完的结果为：把 0000 0104h 地址处的 32 bit 数据放入到 A1 寄存器中，并且 A4 寄存器的内容修改为 0000 0104h。

对某存储空间进行访问时，应首先把此存储空间的地址加载到基址寄存器中，C6000 提供了两条指令来加载存储空间的地址。

例　MVKL *label*，A4

MVKH *label*，A4　(*label* 为变量或 32 bit 无符号常数)

3．指令和功能单元间的映射

C6000 汇编语言的每一条指令只能在一定的功能单元内执行，因此就形成了指令和功能单元之间的映射关系。表 2.63 列出了 C6000 的共享指令到功能单元的映射，表 2.64 列出了 C67xx 的特定指令到功能单元的映射。

表 2.63　C6000 的共享指令到功能单元的映射

.L Unti	.M Unit	.S Unit	.D Uint
ABS	MPY	ADD　SET	ADD　STB(15bit offset)**
ADD	MPYU	ADDK　SHL	ADDAB　STH(15bit offset)**
ADDU	MPYUS	ADD2　SHR	ADDAH　STW(15bit offset)**
AND	MPYSU	AND　SHRU	ADDAW　SUB
CMPEQ	MPYH	B disp　SSHL	LDB　SUBAB
CMPGT	MPYHU	B IRP*　SUB	LDBU　SUBAH
CMPGTU	MPYHUS	B NRP*　SUBU	LDH　SUBAW
CMPLT	MPYHSU	B reg　SUB2	LDHU　ZERO
CMPLTU	MPYHL	CLR　XOR	LDW
LMBD	MPYHLU	EXT　ZERO	LDB(15 bit offset)**
MV	MPYHULS	EXTU	LDBU(15 bit offset)**
NEG	MPYHSLU	MV	LDH(15 bit offset)**
NORM	MPYLH	MVC*	LDHU(15 bit offset)**

续表

.L Unti	.M Unit	.S Unit	.D Uint
NOT	MPYLHU	MVK	LDW(15bit offset)**
OR	MPYLUHS	MVKH	MV
SADD	MPYLSHU	MVKLH	STB
SAT	SMPY	NEG	STH
SSUB	SMPYHL	NOT	STW
SUB	SMPYLH	OR	
SUBU	SMPYH		
SUBC			
XOR			
ZERO			

注：带*的只用于.S2 单元，带**的只用于.D2 单元。

表 2.64　C67xx 的特定指令到功能单元的映射

.L Unit	.M Unit	S. Unit	.D Unit
ADDDP	MPYDP	ABSDP	ADDAD
ADDSP	MPYI	ABSSP	LDDW
DPINT	MPYID	CMPEQDP	
DPSP	MPYSP	CMPEQSP	
DPTRUNC		CMPGTDP	
INTDP		CMPGTSP	
INTDPU		CMPLTDP	
INTSP		CMPLTSP	
INTSPU		RCPDP	
SPINT		RCPSP	
SPTRUNC		RSQRDP	
SUBDP		RSQRSP	
SUBSP		SPDP	

4．延迟间隙

C6000 指令的执行具有延迟间隙(Delay Slots)。延迟间隙数等于从指令的源操作数被读取直到执行的结果可以被访问所使用的指令周期数。对于一个单周期类型指令(如 ADD)，源操作数在第 i 周期被读取，计算结果在第 i+1 周期即可被访问；而对于乘法指令(MPY)，源操作数同样在第 i 周期被读取，计算结果则在第 i+2 周期才能被访问。因此在编写 C6000 汇编程序时一定要特别注意每一条指令的延迟间隙，并不是当前指令就能访问上一条指令的执行结果。

C6000 所有的共享指令都具有一个功能单元等待时间，即每一时钟周期，功能单元都能够重新开始读入一条新指令。单周期功能单元等待时间的另一叫法是单周期吞吐量。而 C67xx 的某些特定指令(包括双精度浮点加法、减法、乘法、比较及 32 位整数乘法指令)的功能单元等待时间都大于 1，即它们占用功能单元的 CPU 周期数大于 1。例如，指令 ADDDP 的功能单元等待时间为 2，即操作数在第 i 和 i+1 周期被读取，则下一条指令只能在第 i+2

周期才开始执行，而非第 i+1 周期。由于该指令的延迟间隙是 6，故计算结果在第 i+7 周期才能被访问。基于以上原因，C67xx 的特定指令不仅要用延迟间隙来予以说明，还要用到功能单元等待时间这个概念。表 2.65 和表 2.66 分别列出了 C6000 的共享指令和 C67xx 各种指令的延迟间隙和功能单元等待时间。

表 2.65　C6000 的共享指令的延迟间隙和功能单元等待时间

指令类型	延迟间隙	功能单元等待时间	读周期*	写周期*	跳转发生*
NOP	0	1			
Store	0	1	i	i	
单周期	0	1	i	i	
乘法(16×16)	1	1	i	i+1	
Load	4	1	i	i, i+4	
跳转	5	1	i**		i+5

注：带*的第 i 周期发生在 E1 节拍；带**的跳转到标号；IRP 和 NRP 的跳转指令不读任何寄存器。

表 2.66　C67xx 指令的延迟间隙和功能单元等待时间

指令类型	延迟间隙	功能单元等待时间	读周期*	写周期*
单周期	0	1	i	i
2 周期 DP	1	1	i	i,i+1
4 周期	3	1	i	i+3
INTDP	4	1	i	i+3, i+4
Load	4	1	i	i, i+4
DP 比较	1	2	i. i+1	i+1
ADDDP / SUBDP	6	2	i, i+1	i+5, i+6
MPYI	8	4	i, i+1, i+2, i+3	i+8
MPYID	9	4	i, i+1, i+2, i+3	i+8, i+9
MPYDP	9	4	i, i+1, i+2, i+3	i+8, i+9

注：带*的第 i 周期发生在 E 节拍。

5．并行操作

取指单元总是一次从存储器中读取 8 条指令，组成一个取指包。取指包的基本格式由图 2.11 给出。取指包必须放在 256 位(8 字)地址边界上(地址的最低 5 位总为 0)。

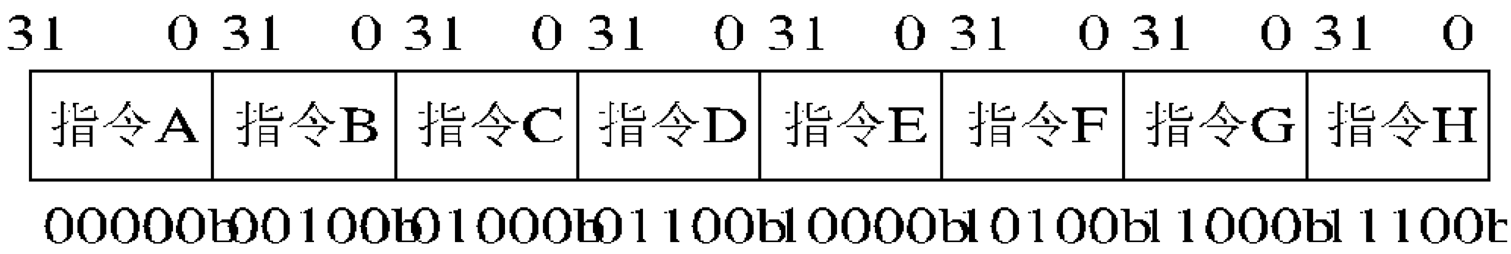


图 2.11　取指包的基本格式

取指包中 8 条指令的执行顺序有三种不同方式：完全串行、完全并行和部分串行。这三种执行顺序的指令代码安排形式如下所示：

完全串行表达形式	完全并行表达形式	部分串行表达形式(例)
指令 A	指令 A	指令 A
指令 B	‖指令 B	指令 B
指令 C	‖指令 C	指令 C
指令 D	‖指令 D	‖指令 D
指令 E	‖指令 E	‖指令 E
指令 F	‖指令 F	指令 F
指令 G	‖指令 G	‖指令 G
指令 H	‖指令 H	‖指令 H

完全串行执行情况下，8 条指令的前面都没有并行符号“‖”， 此指令安排形式导致下列执行顺序(执行包)：

周期 / 执行包	指　令
1	A
2	B
3	C
4	D
5	E
6	F
7	G
8	H

完全并行执行情况下，指令 B 到指令 H 前都有并行符号“‖”，并行符号表示此指令与前一条指令并行执行，此时所有指令使用的功能单元必须各不相同。此指令安排形式导致下列执行顺序(执行包)：

周期 / 执行包	指　　令
1	A　B　C　D　E　F　G　H

部分串行是指部分指令前有并行符号，例中的部分串行指令安排形式导致下列执行顺序(执行包)：

周期 / 执行包	指　令
1	A
2	B
3	C　D　E
4	F　G　H

如果有跳转指令使程序执行过程中由外跳转至一执行包的某一条指令处，则程序从这条指令继续执行，而此执行包中跳转目标之前的所有指令将被忽略。以前面的部分串行例子为例，如果跳转目标是指令 D，则只有指令 D 和 E 将被执行。虽然指令 C 与 D 和 E 处于同一执行包中，它也得不到执行。至于指令 A 和 B，由于处于前一执行包，更不会得到执行。所以如果程序的结果依赖于指令 A、B 或 C 的执行数据，向指令 D 的跳转将会引起错误的结果。

6．条件操作

C6000 的所有指令都可以条件执行。在指令代码中，使用方括号对条件操作进行描述，方括号内是条件寄存器的名称。只有寄存器 B0、B1、B2、A1 和 A2 可作为条件寄存器。

下面例子所示的执行包中含有两条并行的 ADD 指令：第一条 ADD 指令在寄存器 B0 非零时执行；第二条 ADD 指令在 B0 为零时执行。

例
```
      [B0]    ADD   .L1     A1, A2, A3
   || [!B0]   ADD   .L2     B1, B2, B3
```

以上两条指令是相互排斥的，也就是说只有一条指令将会被执行。互斥指令被安排并行时有一定的限制，在下面再详细介绍。

7. 资源限制

在同一执行包中，任何两条指令都不能使用相同的功能单元。在同一指令周期内，不能有两条指令对相同的寄存器执行写操作。下面将较为详细地介绍指令在资源方面的限制。

1) 使用相同功能单元的指令限制

使用相同功能单元的两条指令不能被安排在同一执行包中。

下面的执行包是无效的：

```
     ADD  .S1    A0, A1, A2
   || SHR .S1    A3, 15, A4          ; S1 被两条指令同时使用
```

执行包经过如下改动后成为有效的：

```
     ADD  .L1    A0, A1, A2
   || SHR .S1    A3, 15, A4          ; 使用两个不同的功能单元
```

2) 使用交叉通路(1X 和 2X)的限制

每一执行包中每一数据通路的一个功能单元(L、S 或 M)可以通过交叉通路访问另一数据通路的寄存器组。例如，S1 功能单元可以将某一指令的两个源操作数从寄存器组 A 读出，也可以将其中的一个源操作数从寄存器组 A 读出，而另一源操作数通过交叉通路 1X 从寄存器组 B 读出。目的操作数不能使用交叉通路。交叉通路在语法中的表示是在相应的功能单元符号后加一个“X”后缀。

使用同一条交叉通路的两条指令不能被安排在同一个执行包中，这是因为，从寄存器组 A 到 B 或者从 B 到 A 都只有一条交叉通路。

下面的执行包是无效的：

```
     ADD    .L1X    A0, B1, A1
   || MPY   .M1X    A4, B4, A5          ; 1X 被两条指令同时使用
```

执行包经过如下改动后成为有效的：

```
   ADD      .LIX    A0, B1, A1
   || MPY   .M2X    B4, A4, B2          ; 使用了 1X 和 2X 两条通路
```

3) 数据加载/存储的限制

数据加载/存储所用的地址寄存器必须与所用的功能单元处于同一数据通道中。

下面的执行包是无效的：

```
     LDW    .D1     *A0,A1
   || LDW   .D2     *A2,B2              ; .D2 必须使用组 B 中的寄存器作基址
```

执行包经过如下改动后成为有效的：

```
     LDW    .D1   *A0, A1
   || LDW   .D2   *B0, B2         ; .D 功能单元与基址寄存器处于同一数据通路中
```

加载(或储存)相同寄存器组的两条加载(或储存)指令不能被安排在同一个执行包中；加载和储存相同寄存器组的加载和储存指令也不能被安排在同一个执行包中。

下面的执行包是无效的：

```
   LDW   .D1   *A4, A5
|| STW   .D2   A6, *B4        ; 加载到 A5、储存 A6 处于同一寄存器组
```

执行包经过如下改动后成为有效的：

```
   LDW   .D1   *A4, A5
|| STW   .D2   B6, *B4        ; 加载到 A5、储存 B6 处于不同寄存器组
```

4) 长数据类型(40 位)的限制

因为.S 和.L 单元共用一套为长源操作数另外配置的 8 bit 读口和为长结果另外配置的 8 bit 写口，所以每一执行包中只能允许每一数据通路读/写一个长数据类型。

下面的执行包是无效的：

```
   ADD   .L1   A5:A4, A1, A3:A2
|| SHL   .S1   A8, A9, A7:A6        ; 两个长数据写在同一寄存器组中
```

执行包经过如下改动后成为有效的：

```
   ADD   .L1   A5:A4, A1, A3:A2
|| SHL   .S2   B8, B9, B7:B6        ; 两个长数据不写在同一寄存器组中
```

因为.S 和.L 单元的长数据 8 bit 读口与数据存储通路共用，所以.S 单元或.L 单元的长数据读操作和存储操作不能安排在同一个执行包中。

下面的执行包是无效的：

```
   ADD   .L1   A5:A4, A1, A3:A2
|| STW   .D1   A8, *A9              ; 同时执行长数据读操作和存储操作
```

执行包经过如下改动后成为有效的：

```
   ADD   .L1   A4, A1, A3:A2
|| STW   .D1   A8, *A9              ; 去掉长数据读操作
```

5) 寄存器读限制

对同一寄存器在一个指令周期内读取多于四次是不允许的，条件寄存器不在此限制之列。

下面的执行包是无效的

```
   MPY   .M1   A1, A1, A4
|| ADD   .L1   A1, A1, A5
|| SUB   .D1   A1, A2, A3           ; 对寄存器 A1 进行五次读
```

执行包经过如下改动后成为有效的：

```
   MPY   .M1   A1, A1, A4
|| ADD   .L1   A0, A1, A5
|| SUB   .D1   A1, A2, A3           ; 只对寄存器 A1 进行四次读
```

6) 寄存器写限制

在一个指令周期内，不能同时存在两条指令写入同一寄存器。具有同一目的寄存器的两条指令可以安排并行，只要向该目的寄存器的写操作不在同一个指令周期内发生就可以了。例如，第 i 周期的 MPY 指令与其后第 i+1 周期的 ADD 指令不能写入相同的寄存器，

因为两条指令的写操作都在第 i+1 周期发生。因此，下面的代码序列是无效的，除非在 MPY 指令后有跳转发生，从而阻止 ADD 指令的执行。

```
MPY     .M1    A0, A1, A2
ADD     .L1    A4, A5, A2
```

下面的代码序列却是有效的：

```
MPY     .M1    A0, A1, A2        ；MPY 指令的寄存器写在第 i+1 周期发生
|| ADD  .L1    A4, A5, A2        ；ADD 指令的寄存器写在第 i 周期发生
```

下面的例子是存在寄存器写冲突的代码被汇编器识别的情况：

```
L1:          ADD.L2 B5,B6,B7      ; \汇编器能发现写冲突
||           SUB.S2 B8,B9,B7      ; /
L2:          MPY.M2 B0,B1,B2      ; \汇编器不能发现写冲突
L3:          ADD.L2 B3,B4,B2      ; /
L4: [!B0]    ADD.L2 B5,B6,B7      ; \ 无写冲突
||  [B0]     SUB.S2 B8,B9,B7      ; /
L5: [!B1]    ADD.L2 B5,B6,B7      ; \汇编器不能发现可能存在的写冲突
||  [B0]     SUB.S2 B8,B9,B7      ; /
```

在执行包 L1 中，指令 ADD 与 SUB 写入同一寄存器，这个冲突易被发现。执行包 L2 中的 MPY 指令与 L3 中的 ADD 指令同时写入寄存器 B2，然而，如果存在跳转指令使执行包 L2 之后是其它的执行包而非 L3 的话，则写冲突不会发生，因此 L2 与 L3 之间潜在的写冲突不会被汇编器发现。L4 中的指令不会发生写冲突，因为它们是互斥的。相比之下，L5 中的指令既可能是也可能不是互斥的，汇编器无法判断是否存在写冲突。如果程序执行过程中确实存在写冲突，则结果不确定。

8．线性汇编

TI C6000 汇编优化器的创新之处就是可以把线性汇编代码作为其输入，经优化后生成高效的并行汇编代码。利用线性汇编代码，即使对于不太熟悉 C6000 汇编语言规则的人员来说，也能够快速开发出高效的 C6000 汇编程序，大大缩短了软件开发周期。线性汇编代码类似于前面介绍的 C6000 汇编代码，不同的是线性汇编代码中不需要给出汇编代码必须指出的所有信息，线性汇编代码对这些信息可进行一些选择，或者由汇编优化器确定。采用线性汇编代码不需要考虑如下信息：

- 使用的寄存器；
- 指令的并行与否；
- 指令的延迟间隙；
- 指令使用的功能单元。

如果代码中没有明确指定这些信息，汇编优化器会自动根据代码的情况确定这些信息。与其它代码产生工具一样，有时需要对线性汇编代码进行修改直到性能满意为止。在修改过程中，可能要对线性汇编代码添加更详细的信息，例如指出应该使用哪个类型的功能单元等。

线性汇编文件中必须包含一些汇编优化器伪指令。汇编优化器伪指令用于区分线性汇编代码和正规汇编代码，并为汇编优化器提供其它代码信息。汇编优化器伪指令有以下特性：

- 线性汇编文件的扩展名必须是“.sa”。
- 线性汇编代码应该包括“.cproc”和“.endproc”命令。“.cproc”和“.endproc”命令

限定了让优化器优化的代码段，“.cproc”放在这段代码的开始位置，“endproc”放在这段代码的结尾。

- 线性汇编代码中可能包含“.reg”命令，此命令定义一些符号代表下面指令中用到的寄存器，汇编优化器会自动为这些符号选择一个寄存器。
- 线性汇编代码中可能包括“.trip”命令，此命令限定循环迭代次数。

下例中分别给出了实现两矢量点乘的 C 程序、线性汇编程序以及经汇编优化器优化输出的并行汇编程序。

定点点乘 C 程序

```
int dotp(short a[], short b[] )
{
  int sum0, sum1, sum, i;
  sum0 = 0;
  sum1 = 0;
  for(i=0; i<100; i+=2){
  sum0 += a[i] * b[i];
  sum1 += a[i + 1] * b[i + 1];
  }
  sum = sum0 + sum1;
  return(sum);
     }
```

定点点乘线性汇编程序

```
        .global _dotp
_dotp: .cproc a, b
        .reg sum, sum0, sum1, cntr
        .reg ai_i1, bi_i1, pi, pi1
        MVK 50, cntr   ; cntr = 100/2
        ZERO sum0   ; multiply result = 0
        ZERO sum1   ; multiply result = 0
LOOP: .trip 50
        LDW *a++, ai_i1        ; load ai & ai+1 from memory
        LDW *b++, bi_i1        ; load bi & bi+1 from memory
        MPY ai_i1, bi_i1, pi       ; ai * bi
        MPYH ai_i1, bi_i1, pi1    ; ai+1 * bi+1
        ADD pi, sum0, sum0      ; sum0 += (ai * bi)
        ADD pi1, sum1, sum1     ; sum1 += (ai+1 * bi+1)
        [cntr] SUB cntr, 1, cntr   ; decrement loop counter
        [cntr] B LOOP           ; branch to loop
        ADD sum0, sum1, sum    ; compute final result
        .return sum
        .endproc
```

定点点乘并行汇编代码的核程序

```
     MVK .S1 50, A1  ; set up loop counter
||   ZERO .L1 A7     ; zero out sum0 accumulator
||   ZERO .L2 B7     ; zero out sum1 accumulator
LOOP:
     LDW .D1 *A4++, A2   ; load ai & ai+1 from memory
||   LDW .D2 *B4++, B2    ; load bi & bi+1 from memory
     SUB .S1 A1, 1,A1     ; decrement loop counter
[A1] B .S1 LOOP           ; branch to loop
     NOP 2
     MPY .M1X A2, B2,A6   ; ai * bi
||   MPYH .M2X A2, B2, B6 ; ai+1 * bi+1
     NOP
     ADD .L1 A6, A7, A7 ; sum0+= (ai * bi)
||   ADD .L2 B6, B7, B7 ; sum1+= (ai+1 * bi+1)
; Branch occurs here
    ADD .L1X A7, B7, A4 ; sum = sum0 + sum1
```

2.2.7 TMS320C6000 DSP 的 C/C++语言编程

C/C++编译器的操作方法在第三章关于集成开发环境 CCS 的介绍中再作说明，本节只对有关 TMS320C6000 的 C/C++语言编程的注意问题进行说明。

1．C/C++编译器的输出段

C6000 C/C++编译器的输出段与 C5000 C/C++编译器的输出段基本相同(读者可以参阅 2.1.7 节)，C6000 的 C/C++编译器另外还输出一个.far 段，用来存放声明为 far 的全局和静态变量。

与 C5000 不同，C6000 没有专门的堆栈指针 SP，C6000 的 C/C++程序利用 B15 寄存器作为堆栈指针来管理堆栈。

C6000 的 C/C++编译器支持两种存储器模式：小存储器模式和大存储器模式。

在小存储器模式下(默认)，.bss 段的大小不能超过 32 K 字节，即对程序中的所有全局和静态变量所分配的空间不能超过 32 K 字节。编译器在初始化时就设置好数据页指针 DP(B14 寄存器)，使其指向.bss 段的开始，之后，就可以直接寻址.bss 段中的所有数据，而不用修改数据页指针。

在大存储器模式下，.bss 段的大小不受限制。但是，在访问大模式.bss 段中的全局和静态变量之前，必须先把变量的地址加载到寄存器中。编译器会利用两个额外的指令来加载变量地址：MVKL _x, A0 和 MVKH _x, A0。

下面给出了一个包含 C/C++程序和汇编程序的链接命令文件(*.cmd)例子(以 C6711 为例)。

链接命令文件例子

```
-m example.map        /* 生成一个 map 文件 */
-o example.out        /* 输出可执行文件名*/
   main.obj           /* 第一个 C/C++程序模块 */
   sub.obj            /* 第二个 C/C++程序模块*/
   asm.obj            /* 汇编程序模块*/
-l rts6700.lib        /* 运行时支持库 */
-l matrix.lib         /* 运算库 */

MEMORY
 {
   SRAM : origin = 0x00000000, len = 0x10000
 }
SECTIONS
   {
 .vectors: ALIGN(32) {} > SRAM   ; 用户定义的中断矢量段
 .text : ALIGN(32) {} > SRAM
 .const : ALIGN(8) {} > SRAM
 .data : ALIGN(8) {} > SRAM
 .bss : ALIGN(8) {} > SRAM
 .cinit : ALIGN(4) {} > SRAM
 .stack : ALIGN(8) {} > SRAM
 .far : ALIGN(8) {} > SRAM
 .sysmem: ALIGN(8) {} > SRAM
 .switch: ALIGN(4) {} > SRAM
 .cio : ALIGN(4) {} > SRAM
}
```

2．数据类型

TMS320C6000 的 C/C++语言中定义的数据类型如表 2.67 所示，表中包括每一种数据类型的位长、表示方法和取值范围。

表 2.67　TMS320C6000 的 C/C++语言中的数据类型定义

数据类型	位数	表示方法	数值范围
char, signed char	8	ASCII	- 128～127
unsigned char	8	ASCII	0～255
short	16	补码	- 32 768～32 767
unsigned short	16	原码	0～65 535
int, signed int	32	补码	- 2 147 483 648～2 147 483 647
unsigned int	32	原码	0～4 294 967 295
long, signed long	40	补码	- 549 755 813 888～549 755 813 887
unsigned long	40	原码	0～1 099 511 627 775
float	32	IEEE 32 bit 浮点格式	1.175 494e- 38～3.40 282 346e+38
double, long double	64	IEEE 64 bit 浮点格式	2.225 073 85e- 308～1.797 693 13e+308
pointers	32	原码	0～0xFFFFFFFF
Enum	32	补码	- 2 147 483 648～2 147 483 647

在为变量或常数分配存储空间时，一定要特别注意这些数据类型的位数及其数值范围，否则可能会导致错误结果。

3．在 C/C++程序中调用汇编函数

关于 C/C++程序和汇编程序接口所遵守的原则，请参阅 2.1.7 节。下面针对 TMS320C6000 编译器，介绍如何满足这些原则。

1) 函数名要求

在 C/C++主程序的开头首先将要调用的汇编函数声明为外部函数(例如：extern void sub();),在汇编程序中此函数名前必须有一下画线(例如：_sub)，并用.global 伪指令将其声明为全局符号(例如：.global　_sub)。不被 C 程序访问的其它符号前不要加下画线。

2) 参数传递

C/C++程序在调用汇编函数时，前 10 个输入参数分别放入 A4、B4、A6、B6、A8、B8、A10、B10、A12 和 B12 中；对于 long、double 、long double 类型的输入参数，前 10 个输入参数应分别放入 A5:A4、B5:B4、A7:A6…中，依次类推；其余的输入参数都放入堆栈中，如果输入参数类型的位数小于 int，也作为 int 型放入堆栈，单精度浮点型也作为双精度型放入堆栈。

如果汇编函数返回一个整数、指针或单精度浮点型，结果放入 A4 寄存器中；如果返回一个双精度浮点型或长整型，结果放入 A5:A4 寄存器对中；如果被调汇编函数返回一个结构体，C/C++程序会给此结构体分配空间，并把此结构体的地址放入 A3 中传递给汇编函数，汇编函数在返回前应复制结构体的内容到 A3 指向的存储空间中。

C/C++程序把汇编函数的返回地址放入 B3 寄存器中。B15 寄存器用作堆栈指针(SP)，B14 用作数据页指针(DP)，A15 寄存器用作帧指针(FP)。

3) 寄存器保护

如果被调用的汇编函数中修改了 A10～A15、B3、B10～B15 寄存器，则必须在汇编函数的入口处把这些寄存器压入堆栈进行保护，而在汇编函数返回前应从堆栈弹出这些寄存器，恢复其原值。如果 A3 用于传递结构体参数，汇编函数在修改其内容之前也应该对它保护。如果正常使用堆栈，即压入堆栈的所有内容在函数返回前都弹出，则 B15 寄存器不用保护。对于汇编函数中使用的其它寄存器，C/C++程序在调用函数前自动保护，用户不需要考虑。如果是中断程序，则应保护所有使用的寄存器。

4) 存储区分配及链接命令文件修改

存储区分配及链接命令文件修改在前面已作了介绍，这里不再重述。

下面给出一个在 C 程序中调用汇编函数的例子。在此例子中，C 程序调用汇编函数，而且汇编函数也访问 C 程序中定义的全局变量。

C 语言程序

```
extern int asmfunc(int a); /*声明外部函数*/
int gvar = 4;   /* 定义全局变量*/
void main()
{
  int   i = 5;
  i = asmfunc(i);   /*调用汇编函数*/
 }
```

汇编程序

```
 .global _asmfunc
 .global _gvar
_asmfunc:
   LDW   *+b14(_gvar),A3
   NOP    4
   ADD   a3,a4,a3
   STW   a3,*b14(_gvar)
    MV   a3,a4
    B     b3
    NOP  5
```

其它关于如何在 C/C++程序中嵌入汇编行、C/C++程序和汇编程序中的变量及常数如何互访等操作，与 C5000 编译器的方法类似，读者可参阅 2.1.7 节，这里不再重述。

4．C/C++编译器提供的运行时支持库

TMS320C6000 C/C++编译器提供了如下的运行时支持库：

- rts6200.lib, rts6400.lib 和 rts6700.lib——目标代码库，用于 little－endian 模式；
- rts6200e.lib, rts6400e.lib 和 rts6700e.lib——目标代码库，用于 big－endian 模式；
- rts.src——源代码库，包含 C/C++和汇编源代码函数，上述目标代码库由 rts.src 库编译生成。

TMS320C6000 C/C++编译器的运行时支持库包含如下内容：ANSI C/C++标准库、C I/O 库、向主机操作系统提供 I/O 服务的低级函数、内部(Intrinsic)算术函数、系统启动程序(c_int00)、允许 C/C++程序访问特定指令的函数或宏。如果用户程序中用到这些运行时支持函数，就必须把上述特定的支持库(*.lib)链接到用户目标代码中。

5．C/C++程序初始化

C/C++编译器会在 C/C++主程序 main( )之前加入一个 C 初始化程序模块：c_int00，它为 C/C++程序设置运行环境。用户可以利用复位中断跳转到 c_int00 处。例如，下面的例子就是把中断服务取指包放入到复位中断矢量地址中(利用链接命令文件，把.vectors 段放入到复位矢量地址中)。

```
            .sect ".vectors"
                .ref _c_int00               ; C entry point
            .align    32*8*4               ; must be aligned on 256 word boundary
RESET:                                     ; reset vector
            mvkl _c_int00,b0               ; load destination function address to b0
            mvkh _c_int00,b0
            b b0                           ; start branch to destination function
            mvc PCE1,b0                    ; address of interrupt vectors
            mvc b0,ISTP                    ; set table to point here
            nop 3                          ; fill delay slot
            nop
            nop
```

c_int00 程序完成以下内容：

- 为系统定义一个名为 .stack 的堆栈，并设置堆栈指针和帧指针；
- 将 .cinit 内容拷贝到 .bss 段，对全局和静态变量初始化；
- 设置数据页指针 DP，在小模式下，DP 不会修改；
- 调用 main 开始的 C 程序。

2.3　ADSP2106x DSP 的内部功能结构及源代码开发

2.3.1　ADSP2106x DSP 的功能和结构特点

ADSP2106x 采用超级哈佛结构，其内部功能结构如图 2.12 所示。其特点如下：

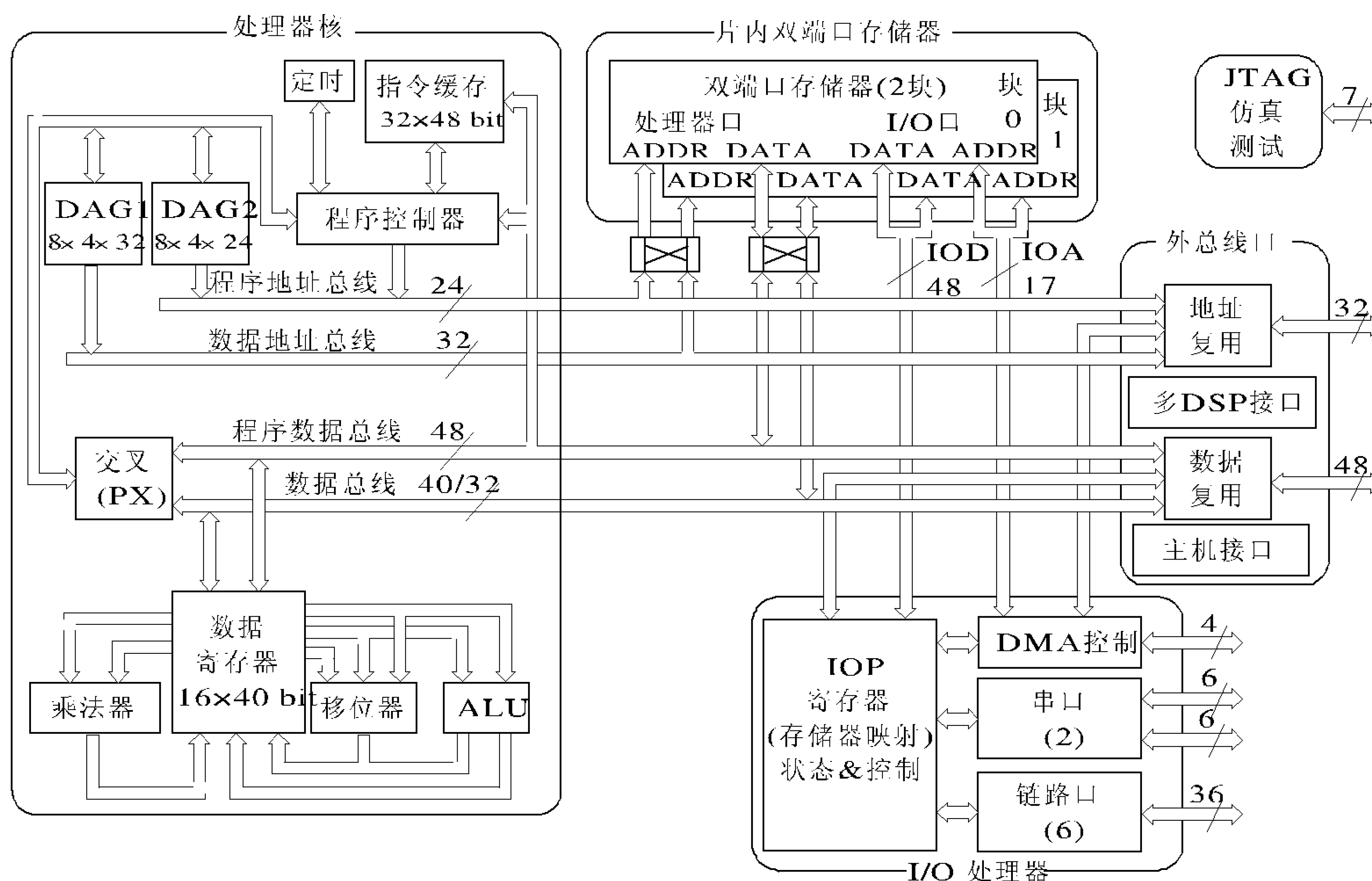


图 2.12　ADSP2106x 的内部功能结构

- 内部有 4 套独立的总线，分别用于双数据存取、指令存取和输入/输出接口。
- CPU 核可以完成 32 bit 定点运算或 32/40 bit 浮点运算，包括乘法器、加法器、移位器在内的计算单元具有 120 MFLOPS 的峰值运算能力，可以在单周期内带条件判断地执行一次乘、一次加、一次减和一次跳转。
- 片内大容量静态存储器(SRAM)分成两块：一块用来存储程序指令及数据，称为程序存储区(PM)；另一块可以用来专门存放数据，称为数据存储区(DM)。这样，如果指令位于缓存(cache)，就可以在单周期内执行乘、加、减运算的同时，分别对 PM 和 DM 区各进行一次数据存取操作。除标准 32 位字宽外，片内存储区还可以灵活地设置成 16 位字宽，以倍增片内存储空间。
- ADSP2106x 有多种外设资源，首先是外部地址、程序/数据总线，可以全速工作在 40 MHz，提供的多种外部控制信号线可以使最多 6 片 ADSP2106x 无需外部控制逻辑就能直接相连，构成一个高效的紧耦合式并行处理系统。另外有六套链路口和两个串行口，用这 6 套链路口可以将大量的 ADSP2106x 构成一个松耦合的并行处理系统。

ADSP21xxx 系列包括了以 ADSP21060 为基础的多个衍生的低成本类型以及早期的 ADSP21020 和新推出的 ADSP21160。表 2.68 中列出了这些芯片的基本配置。

ADSP21060 与 ADSP21062 的内部结构、器件封装、引脚以及指令代码完全一致，只是 ADSP21060 的片内存储器容量是 ADSP21062 的两倍。

表 2.68 ADSP21xxx 系列 DSP 的基本配置

参数 / 型号	外总线/套	片内 SRAM	链路口	DMA 通道	主频(最高) /MHz
ADSP21060	1	4 Mb	6	10	40
ADSP21062	1	2 Mb	6	10	40
ADSP21061	1	1 Mb	无	6	40
ADSP21065	1	512 Mb	无	6	66
ADSP14060	1	4 Mb×4	3×4	10×4	40
ADSP21020	2	无	无	无	33
ADSP21160	1	4 Mb	6	14	100

2.3.2 CPU 核

1. 运算核

ADSP2106x 运算核包含一组数据寄存器和三个独立的计算单元：一个算术逻辑单元(ALU)、一个带定点累加器的乘法器和一个移位器。运算核兼容了早期的 ADSP21020 运算核特点，具备高速、多功能的优点。

ADSP2106x 的所有运算指令只对寄存器操作，因此源操作数必须预装到寄存器中。

ADSP2106x 的运算核除了具备传统 DSP 的加法、乘法、位操作等指令外，还针对数字信号处理常用的运算类型增加了下列指令：简易求倒数、简易求平方根倒数、取两数大、取两数小、位段截取和放置。

将简易求倒数、简易求平方根倒数的结果作为初值(种子)进行迭代运算，可以快速求出高精度的浮点倒数或平方根倒数。ADSP2106x 的算术逻辑单元(ALU)还可以同时求出两个输入数的和与差，这对于快速傅立叶变换(FFT)的核心——碟形运算是十分有用的，再配合以乘法器的并行操作，ADSP2106x 可以在单周期内完成三次浮点操作。

运算核中 16 个 40 bit 寄存器可用于定点或浮点运算。用于定点运算时，只用其中高 32 位进行运算，80 bit 的 MR 寄存器用于定点乘法结果的存放和累加。

ADSP2106x 的计算单元除可以作 32 bit 定/浮点运算外，也可以作 40 bit 浮点的 IEEE 扩展精度运算，定点运算时还可以用 80 bit 累加器进行累加运算。

32 bit 浮点格式符合 IEEE 754/854 标准，并且可以添加 8 个尾数位构成 40 bit 的扩展精度格式。此外，ADSP2106x 还提供 16 bit 的短浮点格式，并能与 32 bit 浮点格式相互转换。浮点运算异常会使 ADSP2106x 的状态标志发生改变，据此可以选择多种处理方法：异常浮点中断，对状态寄存器 ASTAT 和 STKY 进行判断。

浮点处理可以选择 32 bit 或 40 bit。当选择 40 bit 时，浮点运算单元读入 40 bit 数据，并将 40 bit 结果送往 40 bit 宽的寄存器中。当选择 32 bit 时，浮点运算单元接收 32 bit 输入(低 8 bit 置 0)，结果也只取高 32 bit，这与 IEEE 的 32 bit 标准浮点格式是统一的。

算术逻辑单元(ALU)对数据的取整方法有两种：接近 0 方式取整；向最接近的数取整。乘法器只能选择后一种取整方式。ADSP2106x 也可以选择对溢出的定点运算结果采取饱和处理，即当正值溢出时，结果取最大正数：7FFFFFFFh；当负值溢出时，取最大负数：80000000h。如果不设置饱和处理方式，结果直接从高 32 bit 得到。ADSP2106x 运算单元的操作方式(取整、饱和)受 MODE1 寄存器的相应位控制。

80 bit 的定点乘法结果寄存器 MR 比较特殊，从高到低分成 3 个寄存器：MR2(16 bit)、MR1(32 bit)、MR0(32 bit)，它与定点乘法器相连。如果乘法器的两个输入操作数都是小数型定点符号数，乘法器自动将结果左移 1 位以去掉多余的符号位，相乘结果放在 MR1(中 32 bit 寄存器)中。如果乘法器输入为整型定点数，结果放在 MR0(低 32 bit)中。当从 MR2 (16 bit)中读取 32 bit 数据时，高 16 bit 符号扩展。当从 MR2、MR1、MR0 向 40 bit 寄存器送数时，寄存器的低 8 bit 填 0。反之向 MR2、MR1、MR0 写数时，40 bit 寄存器的高 32 bit 被写入。如果是写入到 MR1，则 MR2 作符号扩展；但写入到 MR0 时，不进行符号扩展。ADSP2106x 有两个 MR 寄存器：MRF 和 MRB。

设置 MODE1 能在当前寄存器组 R0～R15 和备用寄存器组(另一套 R0～R15)间切换。运算操作对状态标志的影响记录在寄存器 ASTAT 和 STKY 中。

所有的计算操作都是单周期内完成的，所有的这些计算单元都是并行连接的。任何计算单元的输出在下一指令周期可充当任何计算单元的输入(即不像 TMS320C5000 和 C6000 的那样，存在指令延迟间隙)。在多运算指令中，ALU 和乘法器同时并行工作。

2．控制单元

ADSP2106x 采用流水线方式执行每一条指令，每条指令包括取指、译码、执行三个周期。为消除跳转指令对流水线的影响，ADSP2106x 支持带两级迟延的跳转/调用/返回指令。

程序控制器和循环堆栈指针相结合可以支持最多 6 级的无开销嵌套循环，每层循环都可以单周期退出。

ADSP2106x 对指令缓存(cache)的管理方式与其它 DSP 不同，它不是对要读取的每条指令都进行 cache 地址比较，只有当 PM、DM 总线都用于存取数据时，才进行 cache 是否“命中”的判断，这样就减少了 cache 更新的次数。cache 可以用 MODE2 寄存器设置为禁止(不使用)或冻结(不更新)方式。ADSP2106x 有丰富的条件执行指令。

3．地址产生器和总线

两套地址产生器 DAG1、DAG2 分别指向 PM 区和 DM 区。每套 DAG都配有 8 个地址寄存器 Ix 和 8 个地址修改寄存器 Mx，ADSP2106x 利用这些寄存器来完成存储器寻址。ADSP2106x 还为每套 DAG 提供了 8 对基地址 Bx 和循环长度寄存器 Lx，用以在同一时间段内进行 8 种循环寻址。ADSP2106x 还支持两套 DAG 地址的位反序寻址。

ADSP2106x 的程序只能放在 PM 区中，而数据可以放在 DM 或 PM 区中。

PM 地址总线宽 24 bit，可以访问最多 16 M 的程序/数据混合存储区。PM 数据总线宽 48 bit，用以存放 48 bit 字长的指令，当用来存放数据时，32 bit 单精度浮点数或 32 bit 定点数将放在 48 bit 的高 32 bit 中。DM 地址总线为 32 bit 宽，具有 4 G 寻址空间，DM 数据总线为 40 bit 宽，单精度浮点数或 32 bit 定点数放在其高 32 bit 中。DM 数据总线可以与 DSP 的任何寄存器进行数据传递。还有一个特殊的 PX 总线交换寄存器可以在 48 bit PM 数据总线和 40 bit DM 数据总线或 40 bit 寄存器组之间传递数据。

4．寄存器组成

1) 通用寄存器

• 运算寄存器：包括数据寄存器 R0～R15，当用于浮点运算时书写为 F0～F15；寄存器 MRx，包括 MRF 和 MRB，都可以存定点乘法器结果(80 bit)。MRF 或 MRB 都分为 MR2(高 16 bit)、MR1(中 32 bit)和 MR0(低 32 bit)三个寄存器。

ADSP2106x 的所有运算都是在 R0～R15 和 MRx 寄存器中完成的。

• 程序控制寄存器：包括 PC(程序计数器)、PCSTK(24 bit 的 PC 栈顶地址(最高地址)，30 级硬件堆栈，支持 30 级中断/调用)、PCSTKP(PC 堆栈已用数)、FADDR(取指地址，只读)、DADDR(译码地址，只读)、LADDR(循环 Loop 终止地址，循环地址堆栈的栈顶)、CURLCNTR(当前循环计数器，Loop 计数器堆栈的栈顶)、LCNTR(下一层 Loop 的循环计数值)。

• 地址产生寄存器(DAG1 和 DAG2)：包括 I7～I0 (DAG1 地址寄存器)、M7～M0(DAG1 地址修改寄存器)、L7～L0 (DAG1 循环寻址长度寄存器)、B7～B0 (DAG1 循环寻址基址寄存器)、I15～I8 (DAG2 地址寄存器)、M15～M8 (DAG2 地址修改寄存器)、L15～L8 (DAG2 循环寻址长度寄存器)、B15～B8 (DAG2 循环寻址基址寄存器)。

• 总线交换寄存器：包括 PX2 (32 bit，PM↔DM 总线交换寄存器)、PX1(16 bit，PM↔DM 总线交换寄存器)、PX (48 bit，PX2 高 32 bit 与 PX1 低 16 bit 组合)。

• 定时器：包括 TPERIOD(定时器周期数)、TCOUNT (定时器计数器)。

• 系统寄存器：包括 MODE1(模式控制与状态寄存器 1)、MODE2(模式控制与状态寄存器 2)、IRPTL(中断信号锁存寄存器)、IMASK(中断信号屏蔽/使能寄存器)、IMASKP(中断屏蔽指针，指向嵌套的中断)、ASTAT(运算状态标志，位测试标志)、STKY(辅助运算状态标志、堆栈状态标志)、USTAT1(用户状态寄存器 1)、USTAT2 (用户状态寄存器 2)。

2) IOP 寄存器

IOP 寄存器包括 SYSCON(系统设置寄存器)、SYSTAT(系统状态寄存器)、WAIT(等待寄存器)、VIRPT(多处理器矢量中断寄存器)。

3) 存储器映射寄存器

存储器映射寄存器包括 DMACx(DMA 控制寄存器，x=6,7,8,9)、IIx、IMx、Cx、EIx、EMx、ECx(DMA 参数寄存器，x=0,1,…,9)、STCTLx(串口发送控制寄存器，x=0,1)、SRCTLx(串口接收控制寄存器，x=0,1)、LCTL(链路缓冲控制寄存器)、LCOM(链路口通用控制寄存器)、LAR(链路口指定寄存器)。

对最常使用的寄存器，ADSP2106x 提供了两套同样的寄存器，以便在调用程序和被调用程序中作上下文切换时使用。这样的主寄存器/备用寄存器包括地址产生寄存器(DAG1、DAG2)和数据寄存器 R0～R15。

表 2.69 至表 2.74 列出了模式控制和状态寄存器的位定义。其它寄存器在后文遇到时再作介绍。

表 2.69　模式控制与状态寄存器(MODE1)的位定义

位	名　称	描　　述
0	BR8	I8 的位反序使能，1=使能位反序寻址
1	BR0	I0 的位反序使能，1=使能位反序寻址
2	SRCU	MR 备用寄存器选择，1=选择备用寄存器，0=选择主寄存器
3	SRD1H	DAG1 高 4 寄存器(7～4)备用选择，1=选择备用，0=选择主寄存器
4	SRD1L	DAG1 低 4 寄存器(3～0)备用选择，1=选择备用，0=选择主寄存器
5	SRD2H	DAG2 高 4 寄存器(15～12)备用选择，1=选择备用，0=选择主寄存器
6	SRD2L	DAG2 低 4 寄存器(11～8)备用选择，1=选择备用，0=选择主寄存器
7	SRRFH	寄存器组(R15～8)备用选择，1=选择备用，0=选择主寄存器
9、8	—	保留
10	SRRFL	寄存器组(R7～0)备用选择，1=选择备用，0=选择主寄存器
11	NESTM	中断嵌套使能，1=使能，0=禁止
12	IRPTEN	全局中断使能，1=使能，0=禁止
13	ALUSAT	ALU 饱和使能，1=使能，0=禁止
14	SSE	短字符号扩展使能，1=使能，0=禁止
15	TRUNC	1=浮点直接截断，0=截断到最接近的数
16	RND32	1=把浮点数截取到 32 bit，0=浮点数为 40 bit
18、17	CSEL	待用条件码(00 表示主处理器)
31～19	—	保留

表 2.70　模式控制与状态寄存器(MODE2)的位定义

位	名　称	描　　述
0	IRQ0E	1= $\overline{IRQ0}$ 沿触发，0= $\overline{IRQ0}$ 电平触发
1	IRQ1E	1= $\overline{IRQ1}$ 沿触发，0= $\overline{IRQ1}$ 电平触发
2	IRQ2E	1= $\overline{IRQ2}$ 沿触发，0= $\overline{IRQ2}$ 电平触发
3	—	保留
4	CADIS	指令 cache 禁止，1=禁止，0=使能
5	TIMEN	定时器使能，1=使能，0=禁止
6	BUSLK	外部总线锁定(对多处理器)，1=锁定，0=未锁定
14～7	—	保留
15	FLG0O	FLAG0 输出、输入选择，1=输出，0=输入
16	FLG1O	FLAG1 输出、输入选择，1=输出，0=输入
17	FLG2O	FLAG2 输出、输入选择，1=输出，0=输入
18	FLG3O	FLAG3 输出、输入选择，1=输出，0=输入
19	CAFRZ	指令 cache 冻结，1=冻结，0=允许新指令进入
27～20	—	保留
29、28	—	芯片版本号
31、30	—	处理器 ID 号码

表 2.71　运算状态寄存器(ASTAT)的位定义

位	名　称	描　　述
0	AZ	算术逻辑单元 ALU 为 0 或浮点下溢出，1=ALU 结果为 0，0=ALU 结果非 0；如果 ALU 浮点结果下溢出，就会置位 STKY 的 AUS 位，同时置位 AZ
1	AV	ALU 溢出，1=溢出(浮点上溢出)，0=未溢出；对于定点数同时置位 STKY 的 AOS 位，对于浮点数同时置位 STKY 的 AVS 位
2	AN	ALU 结果，1=结果为负，0=结果为正
3	AC	ALU 定点进位，1=进位
4	AS	ALU 操作数 X 的符号(ABS 和 MANT 指令)，1=负，0=正
5	AI	ALU 浮点无效数操作，1=无效
6	MN	乘法器结果，1=负
7	MV	乘法器溢出，1=溢出(浮点上溢出)；对于浮点数同时置位 STKY 的 MVS 位，对于定点数同时置位 STKY 的 MOS 位
8	MU	乘法器浮点下溢出，1=下溢出，同时置位 STKY 的 MUS 位
9	MI	乘法器无效浮点数操作，1=无效
10	AF	ALU 浮点操作，1=浮点操作，0=定点操作
11	SV	移位器溢出，1=溢出
12	SZ	移位器结果，1=结果为 0
13	SS	移位器输入量符号，1=输入数为负，0=输入数为正
17～14	—	保留
18	BTF	系统寄存器的位测试标志
19	FLG0	FLAG0 值
20	FLG1	FLAG1 值
21	FLG2	FLAG2 值
22	FLG3	FLAG3 值
23	—	保留
31～24	CACC	8 次比较累计位，bit31 存放最后一次比较结果，bit24 存放最早的比较结果，1=X 操作数大于 Y 操作数，0=Y 操作数大于 X 操作数

表 2.72　辅助运算状态寄存器(STKY)的位定义

位	名　称	描　　述
0	AUS	ALU 浮点下溢出
1	AVS	ALU 浮点上溢出
2	AOS	ALU 定点溢出
4、3	—	保留
5	AIS	ALU 浮点无效操作
6	MOS	乘法器定点溢出
7	MVS	乘法器浮点上溢出
8	MUS	乘法器浮点下溢出
9	MIS	乘法器浮点无效数操作
16～10	—	保留
17	CB7S	DAG1 循环缓冲 7 溢出
18	CB15S	DAG2 循环缓冲 15 溢出
20、19	—	保留
21	PCFL	PC 堆栈满
22	PCEM	PC 堆栈空
23	SSOV	状态堆栈满(MODE1 和 ASTAT)
24	SSEM	状态堆栈空
25	LSOV	循环堆栈满(循环地址和计数器)
26	LSEM	循环堆栈空
31～27	—	保留

表 2.73　系统配置寄存器(SYSCON)的位定义

位	名　称	描　　述
0	SRST	软件复位，如果程序置位此位，CPU 响应复位中断(RSTI)
1	BSO	引导方式选择忽略(置 $\overline{BMS}$ 有效)，1=使能引导存储器(利用 $\overline{BMS}$ 选择线)，0=禁止引导存储器
2	IIVT	1=强制把中断矢量表放在内部 20000h 地址处，0=由引导模式指定
3	TWT	指令数据传送格式，1=48 bit 指令，0=32 bit 数据
5、4	HPM	主机打包模式，00 为不打包，01=16→32，10=16→48，11=32→48
6	HMSWF	主机打包次序，1=高位在前，0=低位在前
7	HPFLSH	主机打包状态刷新，1=执行下述操作：清除 SYSTAT 寄存器的 HPS 状态、清除 DMA 通道的请求计数器、清除未打包完的字
8	IMDW0	片内存储块 0 数据字宽，0=32 bit，1=40 bit
9	IMDW1	片内存储块 1 数据字宽，0=32 bit，1=40 bit
10	ADREDY	有效驱动 REDY 方式，1=有效驱动，0=漏极开路
11	BHD	缓冲挂起禁止，1=防止挂起，0=允许挂起
15～12	MSIZE	外部存储器组(Banks)大小，MSIZE＝$\log_2$(32 bit 组大小)- 13
17、16	EBPR	外部总线访问优先权，01=处理器核总线优先，10=IOP 总线优先，00=均等
18	DCPR	DMA 通道 6、7、8、9 优先权，1=循环优先，0=固定次序
27～19	—	保留
28	IMGR	内部存储器分组(网状多处理器连接)
31～29	—	保留

表 2.74 系统状态寄存器(SYSTAT)的位定义

位	名 称	描 述
0	HSTM	主机控制总线，1=主机控制总线，0=主机未控制总线
1	BSYN	总线逻辑同步，1=同步，0=异步
3、2	—	保留
6～4	CRBM	指示当前总线控制者的 ID 码
7	—	保留
10～8	IDC	指示 ADSP2106x 的 ID 管脚(ID2～0)状态
11	—	保留
12	DWPD	直接写内部存储器挂起，1=挂起，0=完成
13	VIPD	矢量中断挂起，1=挂起，0=完成
15、14	HPS	主机打包状态，00=打包完成，01=打包的第一步，10=打包的第二步，11=保留
31～16	—	保留

2.3.3 存储器组织

ADSP2106x 采用多总线结构，存储空间分成三部分：片内、共享和片外。其片内存储器总线的功能强大，配置相当灵活。

1. 存储器总线

以 ADSP21060 具备的 4 Mb 片内 SRAM 为例，它等量分成两块(分别由 DAG1、DAG2 控制)，可以用 16 bit、32 bit、48 bit 形式来访问。其最大存储容量可以定义为 128 K×32 bit、256 K×16 bit、80 K×40 bit 数据或 80 K×48 bit 指令，也可以定义为多种字长的混合存储。ADSP2106x 有三条内部总线与片内存储器相连：PM 总线、DM 总线和 I/O 总线。在一个时钟周期内三条总线都可以对片内存储器访问。

DM 和 PM 地址总线分别由 DAG1 和 DAG2 产生。在每个时钟周期内，每套总线只能访问某块片内 SRAM 一次。当一条指令有两个操作数位于存储器时，把两个操作数分别放在两块片内 SRAM 中，这样可以提高指令的执行速度。这类多存取操作指令在单周期能够完成的条件是：两套总线(DAG1、DAG2)指向的地址分别位于两块片内 SRAM 中，或者有一个指向高速片外存储器；DAG1 不能指向程序所在存储区；当前指令正好位于指令缓存(cache)中。

在通常情况下，取指将在 48 bit 的 PM 总线上执行，仅当指令的某一操作数位于 PM 总线所指存储区时，ADSP2106x 才试图在指令缓存中寻找与该指令匹配的地址，判断其是否已位于缓存中。这多发生于较短的指令循环体中。

PX 是一个 48 bit 的总线交换寄存器，它由两部分组成：PX2 是高 32 bit；PX1 是低 16 bit。PX 在 16 bit↔48 bit，32 bit↔48 bit 的数据/指令传送过程中十分有用。例如 48 bit 指令码以 16 bit 数据格式存放在某存储区中，需要传送到 PM 区以形成可执行代码时，则要读三次 16 bit 数据，分别放在 PX2、PX1 寄存器中，最后通过 PX 寄存器传送到 PM 总线。当仅使用 PX2 寄存器与 PM 总线传送数据时，PM 数据的高 32 bit 数据被有效传送，低 16 bit 为 0。

当使用 PX1 与 PM 总线传送时，PM 总线的高 16 bit 和低 16 bit 都为 0 或不传送，仅中间 16 bit 被传送。当 PX 与 DM 总线传送数据时，如果 DM 总线指向片外存储器，则 48 bit 数据被传送，若 DM 总线指向片内存储器，仅使用 PX 的高 40 bit。PX2 与 DM 总线传送数据时，仅 DM 总线的高 32 bit 被有效传送。PX1 与 DM 总线传送数据时，高 16 bit 和低 8 bit 无效，中间 16 bit 被有效传送。

2. 存储器映射

ADSP2106x 的存储空间分成三部分：片内存储器空间、多处理器存储空间和外部存储器空间。图 2.13 为 ADSP21060 的存储器映射图。

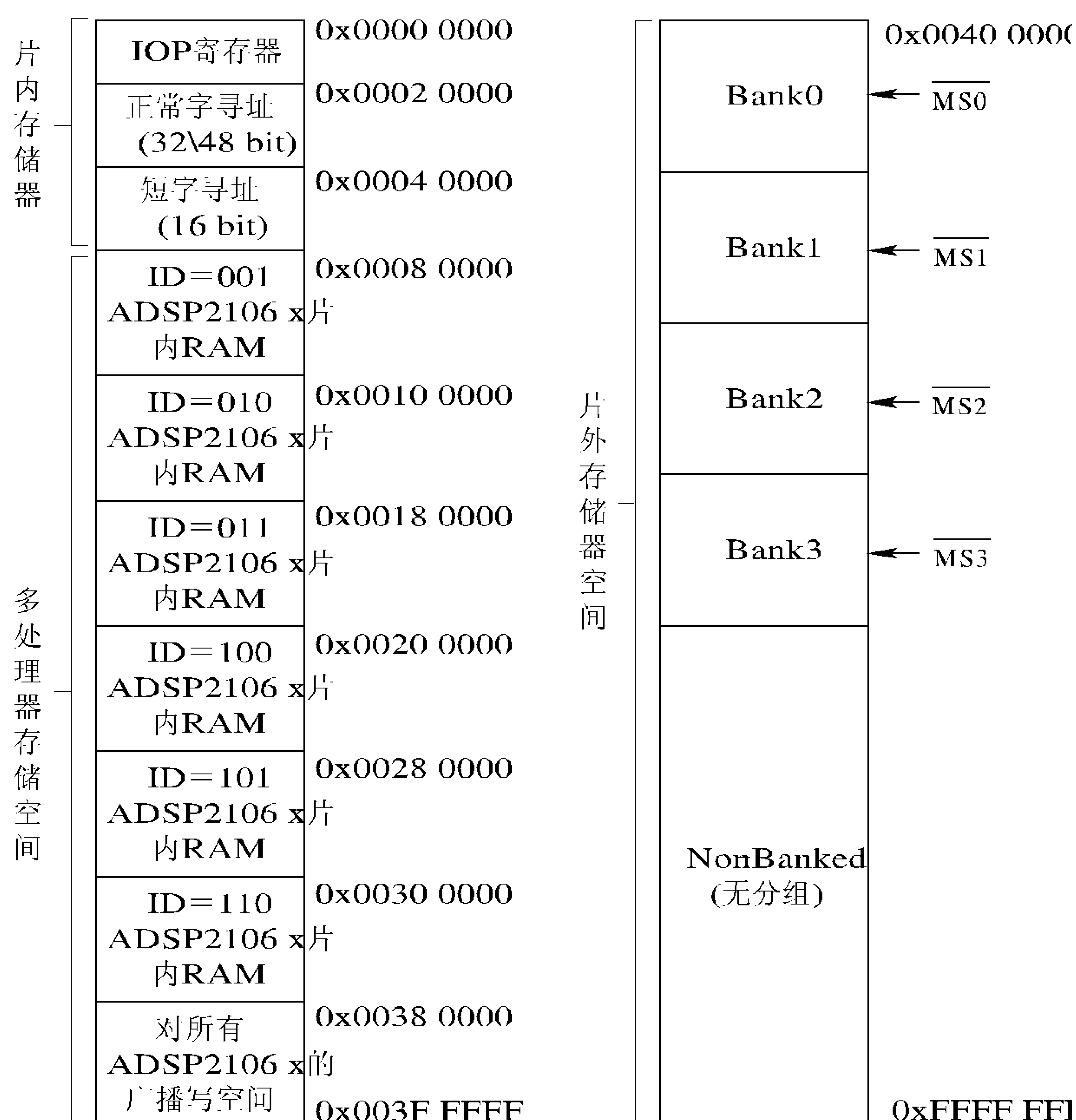


图 2.13　ADSP21060 的存储器映射图

3. 片内存储器

片内存储器地址空间随片内存储器字宽设置的不同而不同。以 ADSP21060 为例，4 Mb 的两个存储块(PM 区和 DM 区)有如下几种情况：

当设置为 32 bit 字宽时，两个存储块 PM 区和 DM 区分别占据 0002 0000h～0002 FFFFh 和 0003 0000h～0003 FFFFh 地址空间。

当设置为 48 bit 字宽时，也分别占据 0002 0000h～0002 FFFFh 和 0003 0000h～0003 FFFFh，但每一存储块末段的部分存储空间不可用。当把一个存储块的前部设置为 48 bit 字(存程序)而后部设置为 32 bit 字(存数据)时，则两部分中间的部分存储空间不可用，这种设置方法常

出现于对 PM 存储块的配置。

当设置为 16 bit 短字格式时，两存储块空间变为 0004 0000h～0005 FFFFh 和 0006 0000h～0007 FFFFh。

事实上，无论设置为 48 bit、32 bit 还是 16 bit 字宽，尽管地址空间不同，但它们都对应同一物理存储空间。

I/O 处理器(简称 IOP)的寄存器是存储器映射寄存器，它们占据 0000 0000h～0000 00FFh 的地址空间范围。IOP 寄存器和正常字地址空间之间的 0000 0100h～0001 FFFFh 为保留地址，不能访问。

4. 多处理器存储共享

ADSP2106x 支持多达 6 片的总线共享连接，并且无需任何外加控制逻辑。每片 DSP 根据其标识码管脚(ID2～0)的电平设置二进制编号：001b～110b(单片独立工作时为 000b)。当每片 DSP 的 PM/DM 总线指向其自身所处的存储空间时，指令/数据的访问表现为片内访问。当指向其它 DSP 所在的存储空间时，表现为多处理器存储共享形式。当指向片外存储器时，则为片外存储器访问。所有共享总线的 ADSP2106x，其外部总线的地址/数据线连在一起，在每个时刻只有一片 ADSP2106x(称主处理器)对此总线有控制权，并可以对其它 ADSP2106x(称从处理器)的片内存储空间甚至控制寄存器进行访问，还可以向其它所有 ADSP2106x 的片内存储器同时写同样内容(广播写)。这一切都靠 ADSP2106x 的片内总线仲裁逻辑和一些多处理器控制握手信号线来完成。当各 ADSP2106x 都访问各自片内存储器或仅有一个 ADSP2106x 访问片外存储器时，各处理器实际上是独立工作的。

5. 片外存储器

ADSP2106x 通过外部总线接口的 32 bit 地址线和 48 bit 数据线以及有关控制信号线访问片外的程序代码或数据，当访问 32 bit 数据时，仅用到 48 bit 数据总线的高 32 bit。DM 总线用 DAG1 产生 32 bit 地址，有 4 G 寻址空间，而 PM 用到的 DAG2 只能寻址包括片内存储器、多处理器寻址空间在内的低 12 M 空间。I/O 总线能访问除 IOP 映射的地址空间和保留空间以外的全部存储空间。外部存储空间分成五个组(Bank)：组 0、组 1、组 2、组 3 和无分组空间，前四组可以用 ADSP2106x 的输出信号线 $\overline{\text{MS0～3}}$ 分别选中，每组存储空间的大小通过 SYSCON 寄存器的 MSIZE 位段来设置。

2.3.4 中断

ADSP2106x 有多种类型的中断：

4 个外部中断：包括复位中断和 3 个外部管脚中断信号；

内部中断包括：定时器中断(2 个)、DMA 控制器中断(10 个)、循环寻址缓冲区溢出中断(2 个)、堆栈溢出(1 个)，运算出错(4 个)，多处理器矢量中断(1 个)，链路服务请求(1 个)及用户定义的软中断(4 个)。

表 2.75 列出了这些中断对应的中断矢量地址和相应的寄存器控制位。复位中断是不可屏蔽的，其余中断都是可屏蔽的，由三个寄存器进行控制。这三个寄存器是：中断锁存标志寄存器(IRPTL)，中断屏蔽寄存器(IMASK)和中断屏蔽寄存器指针(IMASKP)。这三个寄存器分别对各种可屏蔽中断进行标志和屏蔽，在 MODE1 寄存器中还有一个全局中断屏蔽/使

能位 IRPTEN，对所有可屏蔽中断起作用。

表 2.75　ADSP2106x 的中断矢量地址及相应的寄存器(IRPTL，IMASK，IMASKP)控制位

位	矢量地址(Hex)	中断名称	功　能
0	00	—	保留
1	04	RSTI	复位(只读，不可屏蔽，优先级最高)
2	08	—	保留
3	0C	SOVFI	状态 / 循环 / PC 堆栈溢出
4	10	TMZHI	定时器(高优先)
5	14	VIRPTI	矢量中断
6	18	IRQ2I	$\overline{\text{IRQ2}}$ 触发
7	1C	IRQ1I	$\overline{\text{IRQ1}}$ 触发
8	20	IRQ0I	$\overline{\text{IRQ0}}$ 触发
9	24	—	保留
10	28	SPR0I	DMA 通道 0－SPORT0 串口 0 接收
11	2C	SPR1I	DMA 通道 1－SPORT1 串口 1 接收(或 LBUF0)
12	30	SPT0I	DMA 通道 2－SPORT0 串口 0 发送
13	34	SPT1I	DMA 通道 3－SPORT1 串口 1 发送(或 LBUF1)
14	38	LP2I	DMA 通道 4－LBUF2
15	3C	LP3I	DMA 通道 5－LBUF3
16	40	EP0I	DMA 通道 6－EPB0(或 LBUF4)
17	44	EP1I	DMA 通道 7－EPB1(或 LBUF5)
18	48	EP2I	DMA 通道 8－EPB2
19	4C	EP3I	DMA 通道 9－EPB3
20	50	LSRQI	链路服务请求
21	54	CB7I	循环缓冲 7 溢出
22	58	CB15I	循环缓冲 15 溢出
23	5C	TMZLI	定时器(低优先)
24	60	FIXI	定点溢出
25	64	FLTOI	浮点上溢出
26	68	FLTUI	浮点下溢出
27	6C	FLTII	浮点无效
28	70	SFT0I	用户软中断 0
29	74	SFT1I	用户软中断 1
30	78	SFT2I	用户软中断 2
31	7C	SFT3I	用户软中断 3，优先权最低

对于所有外部中断和定时器中断，ADSP2106x 自动将运算状态寄存器(ASTAT)和模式寄存器(MODE1)压入堆栈保存，其压栈操作和中断服务调用并行完成，这样的快速中断服务可嵌套四级。而对于其它寄存器或其它中断服务要保存的寄存器则要用指令来完成压栈操作。

表 2.75 的中断优先级自上而下递降，为了灵活使用定时器，为其定义了两个优先级不同的中断矢量。表中的中断矢量地址是偏移量，当使用片内存储器时，中断矢量基址为 0002 0000h，当使用片外存储器时，基址为 00040 0000h。

2.3.5 片内外设资源

1．直接存储器访问(DMA)

直接存储器访问可以承担数据传输任务而无需 CPU 干预，从而提高了程序的执行效率。ADSP2106x 提供了 10 个 DMA 通道，DMA 控制器能够自动完成不同字宽数据间的打包和展开，并能自动初始化(链式 DMA)。这 10 个 DMA 通道中有 4 个是两种外设接口复用的，如表 2.76 所示。

表 2.76　ADSP21060/62 的 DMA 通道及其控制寄存器、参数寄存器和缓冲寄存器

DMA 通道号	控制寄存器	参数寄存器	数据缓冲寄存器	描　述
0	SRCTL0	II0，IM0，C0，CP0，GP0，DB0，DA0	RX0	串口 0 接收
1	SRCTL1 (LCTL0、LAR、LCOM)	II1，IM1，C1，CP1，GP1，DB1，DA1	RX1 (LBUF0)	串口 1 接收 (链路缓冲 0)
2	STCTL0	II2，IM2，C2，CP2，GP2，DB2，DA2	TX0	串口 0 发送
3	STCTL1 (LCTL0、LAR、LCOM)	II3，IM3，C3，CP3，GP3，DB3，DA3	TX1 (LBUF1)	串口 1 发送 (链路缓冲 1)
4	LCTL0、LAR、LCOM	II4，IM4，C4，CP4，GP4，DB4，DA4	LBUF2	链路缓冲 2
5	LCTL1、LAR、LCOM	II5，IM5，C5，CP5，GP5，DB5，DA5	LBUF3	链路缓冲 3
6	DMAC6 (LCTL1、LAR、LCOM)	II6，IM6，C6，CP6，GP6，EI6，EM6，EC6(DB6，DA6)	EPB0 (LBUF4)	外部口 FIFO 缓冲 0 (链路缓冲 4)
7	DMAC7 (LCTL1、LAR、LCOM)	II7，IM7，C7，CP7，GP7，EI7，EM7，EC7(DB7，DA7)	EPB1 (LBUF5)	外部口 FIFO 缓冲 1 (链路缓冲 5)
8	DMAC8	II8，IM8，C8，CP8，GP8，EI8，EM8	EPB2	外部口 FIFO 缓冲 2
9	DMAC9	II9，IM9，C9，CP9，GP9，EI9，EM9	EPB3	外部口 FIFO 缓冲 3

对于 ADSP21061，只具有除去 LBUF4，LBUF5，EPB2，EPB3 外的 6 个 DMA 通道。

ADSP2106x 还有两对外部 DMA 请求/应答信号线 $\overline{\text{DMAR1}}$/$\overline{\text{DMAG1}}$、$\overline{\text{DMAR2}}$/$\overline{\text{DMAG2}}$，分别对 DMA 通道 7 和通道 8 进行外部握手控制。

DMA 控制器控制着以下 6 种类型的 DMA 传输：内部存储器—外部存储器或外设、内部存储器—内部存储器或其它 ADSP2106x、内部存储器—主机、内部存储器—串口、内部存储器—链路口、外部存储器－外部设备。

1) DMA 寄存器

DMA 寄存器包括控制寄存器、参数寄存器、数据缓冲寄存器和状态寄存器 DMASTAT。

不同 DMA 通道的控制寄存器、参数寄存器、缓冲寄存器有所不同(如表 2.76 所示)。

DMA 控制寄存器用来配置 DMA 通道操作，包括 DMA 使能、链式 DMA(自动初始化)使能、传输方向、数据字宽以及其它一些配置。下面针对不同外设的 DMA 通道再分别介绍。

状态寄存器 DMASTAT 的 bit9～bit0 分别表示了 10 个 DMA 通道的状态：1 表示正在传送数据或等待传送数据，0 笼统表示 DMA 禁止、DMA 完成或正在传送 TCB(链式 DMA 的传送参数块)等多种状态；bit19～bit10 表示各 DMA 通道的 TCB 传送标志：1 表示传送或准备传送 TCB，0 表示链式 DMA 被禁止或未传送 TCB。

DMA 通道的 DMA 参数寄存器如表 2.77 所示。

表 2. 77　DMA 参数寄存器(x 代表 DMA 通道号 0～9)

参数寄存器名	位　数	功　　能
IIx	17	数据源/目的地址
IMx	16	每一数据元素传送完成后的地址调整量(按字宽)
Cx	16	数据传送计数器
CPx	18	链式 DMA 链指针(指向下次 DMA 参数的存放地址)
GPx	17	通用寄存器，也可用于二维 DMA
EIx	32	外部总线地址(仅对外部总线 DMA)
EMx	32	外部总线地址修改量(仅对外部总线 DMA)
ECx	32	外部总线 DMA 传送计数器(仅对外部总线 DMA)
DBx	16	通用寄存器，也可用于二维 DMA(仅对串口/链路口)
DAx	16	通用寄存器，也可用于二维 DMA(仅对串口/链路口)

下面只介绍四个外部口(EPB0～3)DMA 通道的控制寄存器(如表 2.78 所示)，关于串口和链路口 DMA 通道的设置方法，在本节后面对串口和链路口的介绍中再作说明。

表 2. 78　外部口控制寄存器(DMAC6～9)的位定义

位	名　称	功　能　描　述
0	DEN	DMA 使能控制，1=使能，0=禁止
1	CHEN	链式 DMA 使能控制，1=使能，0=禁止
2	TRAN	DMA 方向，1=向外部存储写，0=从外部存储读
4、3	PS	打包状态(只读)，00=打包完成，10=打包或展开的第一步，01=16↔48 bit 或 32↔48 bit 打包 / 展开的第二步
5	DTYPE	数据类型，0=数据，1=指令
7、6	PMODE	打包模式，00=不打包，10＝16↔32，01＝16↔48，11＝32↔48
8	MSWF	打包顺序，1=最高位先打包，0=最低位先打包
9	MASTER	DMA 主机模式，1=是，0=否
10	HSHAKE	DMA 握手模式，1=是，0=否
11	INTIO	单字中断使能，1=使能，0=禁止
12	EXTERN	外设↔外部存储器 DMA，1=是
13	FLSH	1=刷新外部口 FIFO 缓冲
15、14	FS	外部口 FIFO 缓冲状态，00=空，01=半满，11=满
31～16	—	保留

2) 设置DMA的启动步骤

按下列步骤完成DMA通道的数据传送启动：

(1) 清除DMA控制寄存器的DMA使能位(禁止DMA)；

(2) 设置DMA通道的控制寄存器和参数寄存器(源或目的地址IIx、地址调整量IMx及数据元素的计数值Cx等)；(注意，只有一个源或目的地址寄存器，与TI DSP的DMA非常不同。)

(3) 置位DMA控制寄存器中的DMA使能位，这样就启动了DMA。

DMA设置成链式方式时，一旦一个DMA完成，链式DMA将利用预设好的TCB，自动启动DMA链中的下一个DMA。如果要启动一种新的DMA，就必须先禁止DEN位，再设置IIx、IMx、Cx，然后再使能DEN位。有些DMA通道可被两种外设复用(串口/链路、外部总线口/链路)，可通过对相应外设控制寄存器中的DMA使能位设置来选择。如果两种外设控制寄存器中的使能位都置位，将按照串口优先于链路、外部总线口优先于链路来决定DMA通道分配给哪个设备。

3) DMA通道的优先权

10个DMA通道的优先权基本是固定的，从DMAC0到DMAC9依次递降，只有4个外部口DMA之间的优先顺序可以另行设置为循环优先模式，即最新发生DMA传送的那个通道的优先级降到4个外部口DMA通道的最低级，而在复位时，4个外部口DMA通道的优先级从高到低为：通道6、7、8、9。SYSCON寄存器的DCPR位用于设置循环优先模式。

4) 链式DMA

链式DMA使得多个不同的DMA可以自动初始化并依次得到执行。

当链式DMA使能位CHEN置位后，向CP寄存器写入一个非零值就可以启动链式DMA，而向CP写入0则禁止链式DMA。CP寄存器的低17 bit(bit0～16)表示相对于地址0002 0000h的偏移量，bit17则称为可编程控制中断PCI。PCI＝1使得当前DMA完成后产生一个中断请求，如果IMASK寄存器的相应位和MODE1寄存器的全局中断使能位(IRPTEN)使能的话，中断就被响应。

CP寄存器指向链式DMA的下一次DMA的参数寄存器所存放存储区(TCB)的最高地址，存放顺序按地址依次递减为：IIx，IMx，Cx，CPx，GPx，EIx，EMx，ECx。因此建立和启动链式DMA的步骤为：

(1) 在片内存储器建立必要的TCB数据块；

(2) 使能相应的DEN位和CHEN位；

(3) 将TCB的最后一个地址(最高地址)写入CP，即启动了链式DMA。

5) DMA中断

当计数的C寄存器减为0时，将产生DMA中断请求。对于主机模式的外部总线，DMA还要求EC寄存器也减为0才能产生中断请求。要提醒用户的是，向这些C或EC寄存器写入0并不会有中断请求产生。10个DMA通道中断的优先级从DMA通道0到通道9递降。

每个DMA中断请求将锁存在IRPTL寄存器中并由IMASK控制是否使能。对链式DMA来说，CP寄存器的PCI位也控制着中断的产生。

MODE1寄存器的IRPTEN位控制着全局中断使能。

6) DMA 的产生和终止

DMA 在链式/非链式模式下的启动条件不同，当下述条件之一满足时，DMA 将启动：

- 链式 DMA 禁止时，DMA 使能位 DEN 从 0 变为 1，便启动非链式 DMA。
- 链式 DMA 使能且 DEN＝1 时，向 CP 寄存器写入一个非 0 值，便启动链式 DMA。
- 链式 DMA 使能，CP 为非 0，而当前 DMA 结束，链式 DMA 重新开始。

满足下列条件之一，DMA 将终止：

- 计数寄存器 C 和 EC 减为 0。
- 链式 DMA 被禁止，DEN 从 1 变为 0。当 DEN 为 0，且链式 DMA 使能时，通道进入链插入模式。

7) 外部口 DMA 的特别用法

4 个外部口 DMA 通道实际上与 4 个外部端口数据 FIFO 缓存相连，即 DMA 通道 6、7、8、9 分别对应 EPB0、1、2、3。每个 EPBx 缓存有 6 级 FIFO，通向两端口：一个读端口和一个写端口。但当 DMA 使用 EPBx 时，处理器核不应访问此 EPBx。通过向 DMACx 寄存器的 FLSH 位写 1 可以刷新此 EPBx(清空 FIFO)。

DMA 控制寄存器(DMACx)的 TRAN 位可以设置 16 bit、32 bit、48 bit 之间的打包/展开模式。设置 MSMF 位可以确定低位字段先打包还是高位字段先打包，当 32 bit 打包成 48 bit 数据时，需要三次传送，每次 32 位，结果为 2 个 48 bit。

与外部总线 DMA 通道 7、8 相关的有两套 DMA 控制信号线：$\overline{\text{DMAR1/2}}$(请求)和 $\overline{\text{DMAG1/2}}$(确认)。DMA 控制寄存器 DMACx 的 MASTER 位、HSHAKE 位、EXTERN 位设置 DMA 方式，如表 2.79 所示。

表 2.79　外部口 DMA 方式

M	H	E	DMA 运行模式
0	0	0	从属模式，当 DMA 接收缓冲或发送缓冲不满时，产生 DMA 请求
0	0	1	保留
0	1	0	握手模式，当 $\overline{\text{DMARx}}$ 有效时产生 DMA 请求，当 $\overline{\text{DMAGx}}$ 有效时，进行 DMA 传送
0	1	1	外部握手模式，同上，但数据传送发生在外存储器和外部设备之间
1	0	0	主机模式，一旦接收缓冲或发送缓冲不满，DMA 控制器就试图传送。如果 DMA 通道 7 为主机模式，$\overline{\text{DMAR1}}$ 就应恒为高；如果 DMA 通道 8 为主机模式，$\overline{\text{DMAR2}}$ 就应恒为高
1	0	1	保留
1	1	0	协调单步方式，每个 DMA 传送都由 $\overline{\text{DMARx}}$ 来驱动
1	1	1	保留

8) 二维 DMA

DMA 通道 0～5 支持二维 DMA，通过设置相应串口 DMA 控制寄存器的 D2DMA 位或链路 LCOM 控制寄存器的 L2DDMA 位就可以设置成二维 DMA 方式，即以行主模式(先 X 方向，后 Y 方向)访问一个二维阵列元素，这样可以灵活实现循环寻址和数据重排。这时 DMA 通道参数寄存器的功能有所改变，如表 2.80 所示。DMA 通道 1、3、5 固定为发送数据，而 DMA 通道 0、2、4 固定为接收。

表 2.80　二维 DMA 的参数寄存器

寄存器	功　　能
IIx	地址，初始值为阵列起始地址，每次传送加 X 增量(当 X 计数为 0 时，加 Y 增量)
IMx	X 增量，每次传送后在 X 方向的增加值
Cx	X 计数，初始值为 X 初始计数值，每次传送减 1
CPx	下一指针，指向下一个 DMA 参数所存放的内部存储器地址
DBx	Y 增量，当 X 计数减为 0 时，把此值加到地址 IIx 上
GPx	Y 计数，Y 方向的元素个数，相当于行数，每行传送完后(Cx → 0)减 1
DAx	X 初始计数值，二维阵列在 X 方向的元素个数，当 X 计数(Cx)值减为 0 时，DAx 值就重新装入 Cx 计数器中

当 Y 计数和 X 计数都减为 0 时，二维 DMA 结束。

2．多处理器共享存储总线

ADSP2106x 提供了必要的控制握手信号线，使 6 片 ADSP2106x 无需任何外部逻辑就可以直接相连组成一个紧耦合的多处理器系统。在这样一个多处理器系统中，所有 ADSP2106x 的外部总线都连接在一起(或可能相连)。

每个 ADSP2106x 的片内存储器和 IOP 寄存器可以被其它处理器读写。多片共享总线处理器在任一时刻只有一片对外部总线有控制权，称其为主处理器，而其它均为从处理器，其处理器寻址空间被统一映射，即每个处理器的片内存储器将根据 ID2～0 管脚状态被惟一地映射到一段存储地址内。ID 号与处理器一一对应，每个处理器使用总线请求信号 $\overline{\text{BR6～1}}$ 中的一条来作为它使用外部总线的请求，并根据固定优先或循环优先机制来获取总线控制权而成为主处理器。

主处理器不仅可以访问片外共享存储器，还可以访问所有从处理器的片内存储器、IOP 寄存器(也映射成存储器)，并在从处理器上建立 DMA 传送。对从处理器的访问是通过读写($\overline{\text{RD}}$，$\overline{\text{WR}}$)和确认(ACK)等信号握手完成的。主处理器还可以向所有从处理器作广播式数据传送，在存储器映射图(图 2.13)中有一块存储区称为广播式数据区，向它写入数据等价于向所有处理器同时写入此数据，而确认信号 ACK 则由所有从处理器“相与”合成。

ADSP2106x 的片内存储器常常足以放下各处理器自己的指令和局部数据，而它大多数时间只是对各自指令和局部数据访问，使得外部共享总线的访问率大大降低，总线瓶颈的影响对于 ADSP2106x 来说也大大减少；另一方面，总线的传输带宽远远大于链路口的带宽，而且链路传送因接收双方都需初始化而附有较多开销；利用总线还可以作广播式传送，主处理器还能“全面”控制从处理的片内资源。这些因素构成了 ADSP2106x 共享多处理器的优越性。

ADSP2106x 的多处理器共享总线无需片外逻辑控制，处理器自身提供了必要的逻辑仲裁机制。每个处理器根据其 ID 号将 $\overline{\text{BR6～1}}$ 中的一条作为输出来提出总线请求，其它 $\overline{\text{BRx}}$

信号线都作为输入。RPBA 是仲裁优先权选择，RPBA=0 表示按固定次序的优先权，即 ID1 处理器发出的 $\overline{BR1}$ 请求优先权最高，而 ID6 处理器发出的 $\overline{BR6}$ 优先权最低；RPBA=1 表示循环优先方式，即 ID 号跟随当前主处理器的从处理器的优先权最高。$\overline{CPA}$ 是处理器核(core)的优先访问权。$\overline{CPA}$ =0 使得从处理器核可以打断后台 DMA 传送而获得对外部总线的访问权。$\overline{CPA}$ 信号是漏极开路输出的，如果使用这种方式则所有处理器的 $\overline{CPA}$ 应连在一起。

当某从处理器需要控制外部总线(即成为主处理器)时，它在时钟周期的开始将其对应的 $\overline{BRx}$ 信号线置低，并在同一时钟周期的稍后时间采样其它 $\overline{BRx}$ 信号。主处理器通过保持其对应的 $\overline{BRx}$ 有效来维持其总线控制权，当其 $\overline{BRx}$ 无效而并没有从处理器提出总线请求时，主处理的控制权仍不变。只有主处理器的 $\overline{BRx}$ 无效，而某从处理器 $\overline{BRx}$ 有效时，总线控制权的转移才会发生。每个处理器通过采样 $\overline{BRx}$ 信号线就可以知道哪一个将成为新的主处理器，并将这个信息记录在反映新主处理器的 SYSTAT 寄存器的 CRBM 位中。

伴随着总线权的转移，原主处理器使其外部总线成为三态，即 DATA47～0，ADDR31～0，ADRCLK、$\overline{RD}$、$\overline{WR}$、$\overline{MS3\sim0}$、PAGE、$\overline{HBG}$、$\overline{DMAG2\sim1}$ 等信号线在总线权转移的时钟周期末变为三态，而在下一周期初新的主处理器将驱动这些信号。

在总线权转移的周期内，对外部总线的访问将被迟延。

主处理器通过外部总线可以直接读写从处理器的片内存储器和存储器映射的 IOP，这对从处理器核来说是不可见的，因此从处理器核可以继续执行指令。

主处理器通过修改从处理器的 IOP 寄存器，可以产生矢量中断或建立 DMA 传送。

广播式写是主处理器向所有从处理器的同一个片内存储位置或 IOP 寄存器同时写入数据。如果这种广播式写不是 DMA 写，则主处理器也向它自身相应的存储位置写同样的数据。

3．主机接口

主机接口使 ADSP2106x 可以与标准的 16 bit 或 32 bit 总线相连，以同步或异步方式传送数据，主机接口映射在统一的存储空间内，可以使用 4 个外部口 DMA 通道。

$\overline{HBR}$、$\overline{HBG}$、REDY、$\overline{CS}$ 信号线用于主机对 ADSP2106x 外部总线的请求和控制。一旦主机获得了总线控制权，就可以对 ADSP2106x 的内部存储器直接读写。主机用某些 IOP 寄存器如 SYSCON、SYSTAT 来控制 ADSP2106x 或者建立 DMA，也可使用矢量中断 VIRPT。DMA 传送是由 ADSP2106x 的片内 DMA 控制器进行控制的。主机与各处理器的这些作用方式与前面介绍的主、从处理器的作用方式十分类似。

4．链路口

ADSP2106x 提供了 6 个链路口，每个链路口包括 4 位数据线，一个双向时钟信号，一个双向确认信号。每个链路口还可以按 1 倍、2 倍时钟频率的速度通信。链路口有以下特点和功能：

- 各链路可以独立地、同时地工作。
- 链路数据可以打包成 32 bit、48 bit 数据，可以被处理器核访问，可以与片内存储器作 DMA 传送。
- 外部主机可以直接访问链路。

• 发送和接收有双倍缓冲寄存器。

• 通过时钟/确认信号在链路通信时握手，每个链路均可收/发并分别由一个 DMA 通道支持。

• 链路连接可以组成一维到多维的各种形式的处理器网络。

1) 链路口寄存器

链路口寄存器包括链路口指定寄存器(LAR)、链路控制寄存器(LCTL)、链路通用控制寄存器(LCOM)和链路服务请求寄存器(LSRQ)，这些寄存器的位定义分别如表 2.81～表 2.84 所示。

表 2.81　链路口指定寄存器的位定义

位　段	名　称	功　能	备　注	
2～0	A0LB	LBUF0 的指定链路口	链路口编码	链路口
5～3	A1LB	LBUF1 的指定链路口	0 0 0	链路口 0
8～6	A2LB	LBUF2 的指定链路口	0 0 1	链路口 1
11～9	A3LB	LBUF3 的指定链路口	0 1 0	链路口 2
14～12	A4LB	LBUF4 的指定链路口	0 1 1	链路口 3
17～15	A5LB	LBUF5 的指定链路口	1 0 0	链路口 4
31～18	—	保留	1 0 1	链路口 5
			1 1 0	保留
			1 1 1	此 LBUF 未用

注：LAR 寄存器的复位初始值为 2C688h。

表 2.82　链路控制寄存器的位定义

位　段	名　称	功　能	备　注
3～0	*	LBUF 0 控制	* LBUFx 控制(x=0～5)
7～4	*	LBUF 1 控制	位　名称　功能
11～8	*	LBUF 2 控制	0+4x　LxEN　LBUFx 使能
15～12	*	LBUF 3 控制	1+4x　LxDEN　LBUFx DMA 使能
19～16	*	LBUF 4 控制	2+4x　LxCHEN　LBUFx 链式 DMA 使能
23～20	*	LBUF 5 控制	3+4x　LxTRAN　LBUFx 方向，1=发送，0=接收
24	LEXT0	字扩展传送，1=48 bit，0=32 bit	
25	LEXT1	字扩展传送，1=48 bit，0=32 bit	
26	LEXT2	字扩展传送，1=48 bit，0=32 bit	
27	LEXT3	字扩展传送，1=48 bit，0=32 bit	
28	LEXT4	字扩展传送，1=48 bit，0=32 bit	
29	LEXT5	字扩展传送，1=48 bit，0=32 bit	
31、30	—	保留	

表 2.83　链路通用控制寄存器的位定义

位	名　称	功　　能
1～0	L0STAT(1～0)	LBUF0 状态：11=满，00=空，10=1 个字，01=保留
3～2	L1STAT(1～0)	LBUF1 状态：11=满，00=空，10=1 个字，01=保留
5～4	L2STAT(1～0)	LBUF2 状态：11=满，00=空，10=1 个字，01=保留
7～6	L3STAT(1～0)	LBUF3 状态：11=满，00=空，10=1 个字，01=保留
9～8	L4STAT(1～0)	LBUF4 状态：11=满，00=空，10=1 个字，01=保留
11～10	L5STAT(1～0)	LBUF5 状态：11=满，00=空，10=1 个字，01=保留
12	LCLKX20	LBUF0 的发送数据时钟频率倍数，1=2 倍，0=1 倍
13	LCLKX21	LBUF1 的发送数据时钟频率倍数，1=2 倍，0=1 倍
14	LCLKX22	LBUF2 的发送数据时钟频率倍数，1=2 倍，0=1 倍
15	LCLKX23	LBUF3 的发送数据时钟频率倍数，1=2 倍，0=1 倍
16	LCLKX24	LBUF4 的发送数据时钟频率倍数，1=2 倍，0=1 倍
17	LCLKX25	LBUF5 的发送数据时钟频率倍数，1=2 倍，0=1 倍
18	L2DDMA	使能二维 DMA：1=使能，0=禁止
19	LPDRD	禁止对 LxCLK 和 LxACK 的内部下拉电阻：1=禁止，0=使能
20	LMSP	使能多处理器网络
22～21	LPATHD	多处理器网络路径 LPATH 变动迟延：00=无迟延，01=1 个附加迟延，10=2 个附加迟延，11=3 个附加迟延
25～23	—	保留
26	LRERR0	LBUF0 接收出错状态：1=未完成，0=完成
27	LRERR1	LBUF1 接收出错状态：1=未完成，0=完成
28	LRERR2	LBUF2 接收出错状态：1=未完成，0=完成
29	LRERR3	LBUF3 接收出错状态：1=未完成，0=完成
30	LRERR4	LBUF4 接收出错状态：1=未完成，0=完成
31	LRERR5	LBUF5 接收出错状态：1=未完成，0=完成

表 2.84　链路服务请求寄存器的位定义

位	名　称	功　　能	备　　注
3～0	—	保留	
4	L0TM	链路 0 发送屏蔽	
5	L0RM	链路 0 接收屏蔽	
6	L1TM	链路 1 发送屏蔽	
7	L1RM	链路 1 接收屏蔽	
8	L2TM	链路 2 发送屏蔽	
9	L2RM	链路 2 接收屏蔽	
10	L3TM	链路 3 发送屏蔽	
11	L3RM	链路 3 接收屏蔽	
12	L4TM	链路 4 发送屏蔽	
13	L4RM	链路 4 接收屏蔽	
14	L5TM	链路 5 发送屏蔽	
15	L5RM	链路 5 接收屏蔽	
19～16	—	保留	
20	L0TRQ	链路 0 发送请求状态(只读)	LxTRQ=1 相当于 LxACK=1,LxTM=1,LxEN=0
21	L0RRQ	链路 0 接收请求状态(只读)	LxRRQ=1 相当于 LxCLK=1,LxRM=1,LxEN=0
22	L1TRQ	链路 1 发送请求状态(只读)	
23	L1RRQ	链路 1 接收请求状态(只读)	
24	L2TRQ	链路 2 发送请求状态(只读)	
25	L2RRQ	链路 2 接收请求状态(只读)	
26	L3TRQ	链路 3 发送请求状态(只读)	
27	L3RRQ	链路 3 接收请求状态(只读)	
28	L4TRQ	链路 4 发送请求状态(只读)	
29	L4RRQ	链路 4 接收请求状态(只读)	
30	L5TRQ	链路 5 发送请求状态(只读)	
31	L5RRQ	链路 5 接收请求状态(只读)	

ADSP2106x 有 6 个独立的链路缓冲 LBUF5～0，可以自由地定义与 6 个链路口 LINK5～0 的连接对应关系，LBUF 在 DMA 控制下完成与片内存储器之间的数据传送。链路指定寄存器(LAR)用来确定 LBUF5～0 与 LINK5～0 间的连接关系，存储器之间传送数据时可以把一个 LINK 赋予两个 LBUF。LBUF5～0 与 6 个 DMA 通道的对应关系如表 2.76 所示。

链路口在复位后还可用于处理器引导(LBUF4)。

链路控制寄存器有 3 个：链路控制寄存器 LCTL、链路通用控制寄存器 LCOM 和链路指定寄存器 LAR。设置链路操作时，应按照 LAR、LCOM、LCTL 的次序设置这些寄存器。在对 LAR 重新指定前，必须用 LCTL 禁止此 LBUF。

LAR 寄存器初始化值为 2C688h，即链路口 0 指定给 LBUF0，链路口 1 指定给 LBUF1，依此类推。某链路口在两种情况下是禁止的，即没有指定 LBUF，或者指定的 LBUF 被禁止。当链路口禁止时，其数据线 LxDAT3～0 及 LxACK、LxCLK 都是三态。将一链路口指定到两个 LBUF 并禁止这两个 LBUF，就可以用“白环”模式在存储器与存储器之间传送数据，这时 LxDAT3～0、LxACK、LxCLK 都不被驱动。

2) 握手信号

链路握手信号依靠 LxACK 和 LxCLK，链路口以 4 位码一组的方式传送 32 bit 或 48 bit 字(高位在先)，发送方在时钟 LxCLK 的上升沿送出 4 位码，接收方利用时钟下降沿锁存 4 位码，接收方使 LxACK 有效表示已准备好接收下一个字。在每个字开始发送时，发送方如果看到 LxACK 无效，将使 LxCLK 为高并等待 LxACK 有效后才开始发送新字。当发送缓冲器为空时，LxCLK 将保持低。当一个高优先权的 DMA 出现或 DMA 链加载发生时，接收缓冲将填充，这时接收方提前使 LxACK 无效，当缓冲有空位置后再使 LxACK 有效。

发送方驱动 LxDAT3～0、LxCLK，接收方驱动 LxACK。为了允许收/发双方被使能的时间上有先有后，在链路口禁止时对 LPDRD 清 0，来使 LxACK、LxCLK、LxDAT3～0 被内部下拉(50 kΩ)。这 6 条信号线如果悬空，则必须用内部或外部下拉电阻。

3) 链路缓冲 LBUF

每个 LBUF 由一个内部寄存器和一个外部寄存器组成的 2 级 FIFO 构成。在 DMA 方式下，当 LBUF 用于发送时，内部寄存器接收 DMA 从片内存储器送来的数据，外部寄存器将数据字展开成 4 位码，最高位先发送，当 DMA 或处理器核送来的数据占满这 2 级 FIFO 时，将送出一个“满”标志，每当一个字展开发送后，FIFO 中将空出一个位置并发出一个请求，当 FIFO 空时，LxCLK 就无效。同样当 LBUF 用于接收时，外部寄存器用于数据打包，然后数据经内部寄存器以 DMA 方式送入片内存储器。

处理器核可对 LBUF 直接读写，这种访问 LBUF 的方式比 DMA 方式的迟延要小。处理器核通过查询 LCOM 寄存器获知 LBUT 的满/空状态来访问 LBUF，这时 DMA 应禁止。当处理器核读一个空的接收 LBUF 时，它将处于等待状态，直到接收字的到来。类似地，向满的发送 LBUF 写数据也会被挂起，挂起使 DSP 程序停顿。为防止这类挂起可以置位 SYSCON 寄存器的 BHD 位。

主机也可以直接访问 LBUF，其字宽仅由 SYSCON 寄存器的 HPM 位决定，而与 LCTL 寄存器的 LEXT 位无关。

4) 链路 DMA

LBUF0、LBUF1、LBUF4、LBUF5 都与其它 4 种外设共享 DMA 通道，其它 4 种外设具有更高的优先权，如串口 1 接收与 LBUF0 共享 DMA 的通道 1，当串口 1 接收 DMA 使能的同时，LBUF0 DMA 也使能，则串口 1 占用 DMA 通道 1。

链路 DMA 的设置步骤为：

(1) 首先利用 LCTL 寄存器的 LxEN 位禁止(清 0)下面使用的 LBUFx；

(2) 由 LAR 寄存器的 AxLB 位段为链路口指定一个 LBUFx；

(3) 由 LCTL 寄存器的 LxEN 位使能(置位)此 LBUFx，并分别由 LxEXT 和 LxTRAN 位指定传送字宽和方向；

(4) 设置 DMA 通道参数(IIx、IMx 和 Cx)和控制寄存器的其它所需位；

(5) 置位 LCTL 寄存器的 LxDEN 位后，就启动了链路 DMA。

5) 链路口中断

链路口有三种中断形式：DMA 使能时，DMA 完成时将产生一个可屏蔽中断；DMA 禁止时，处理器核可以对存储器映射的 LBUF 进行读写，当接收缓冲不空或发送缓冲不满时就可产生可屏蔽中断；当外部设备访问一个未指定 LBUF 的链路口时，或者访问一个已指定但此 LBUF 被禁止的链路口时，将产生可屏蔽的 LSRQ 中断。

前两种类型使用同一种中断——链路中断，最后一种类型则是另一种中断矢量 LSRQ。

某链路口的 LBUF DMA 禁止时，处理器核可以按存储器映射地址访问此 LBUF。当 LBUF 的接收缓冲非空或发送缓冲不满时产生一个可屏蔽中断，这一中断锁存在 IRPTL 寄存器的相应位中，并可由 IMASK 寄存器的相应位屏蔽。中断服务程序在对 LBUF 读/写后应检查其空/满状态，然后决定是否返回，这样可以减少中断次数。

如果 LBUF DMA 使能，则当 DMA 完成(DMA 计数寄存器减为 0)时将产生中断。因为 DMA 完成时，LBUF 的接收缓冲为空，发送方的处理器可以利用接收缓冲的两个 FIFO 空位置再发两字的额外信息，接收方的处理器中断服务程序可以读出并根据这两个字决定后续操作。

链路中断服务请求 LSRQ 是当链路口未指定或指定的 LBUF 被禁止时，外部设备试图访问此链路口而引起的中断。ADSP2106x 可以根据 LSRQ 寄存器来判断是哪个链路口被何种形式访问。当 LxACK 或 LxCLK 被外部设备置为有效(即试图访问此链路口时)时，就会产生一个链路服务请求 LSR，但一个处于“自环”模式的链路口不会产生 LSR。每个 LSR 在被锁存到 LSRQ 寄存器之前先要经过屏蔽“过滤”，然后 6 个可能的发送 LSR 和 6 个可能的接收 LSR“相或”产生链路服务中断请求 LSRQ，LSRQ 能否得到响应还取决于 IMASK 寄存器的 LSRQI 位。

中断服务程序必须读 LSRQ 寄存器来判断是哪个链路口的何种服务请求，方法是将 LSRQ 读入寄存器 Rx，用检测左起 0 值位个数的指令 Rn=LEFTZ Rx 来确定哪个链路口有服务请求(按优先级)。当 LSR 正在使用时，应在对它们作使能、禁止、指定等操作前将它们屏蔽，以免引起不需要的 LSR。

6) 传送错误检测

LCOM 寄存器的 LRERRx 位反映了在传输 32 bit 或 48 bit 字时，4 位码传送可能出现的错误，即传送 4 位码的个数不是 8 或 12 的整数倍。

5．串行口

ADSP2106x 有两个独立的同步串行口 SPORT0 和 SPORT1，其功能特征有：独立的收/发功能；可发送字宽为 3～32 bit 的数据，可以选择低位先发或高位先发方式；双缓冲，即数据缓冲寄存器和移位寄存器；硬件的 A 律/μ 律压扩功能；串行时钟和帧同步可以由内部产生或由外部输入，频率范围最高可达到核时钟频率；在处理器核控制下，可由中断触发串口与片内存储器间的单字传送；与片内存储器进行 DMA 传送；以链式 DMA 完成多块数据传输；TDM 接口的多通道模式，时分多址数据收发，最多 32 个通道。

每个串行通信口有 6 根信号线：发送数据线(DT0、DT1)、发送时钟线(TCLK0、TCLK1)、发送帧同步线(TFS0、TFS1)、接收数据线(DR0、DR1)、接收时钟线(RCLK0、RCLK1)和接收帧同步线(RFS0、RFS1)。

每个串口可以同时进行收/发双工工作。对每个发送数据位，串口都要根据 TDIVx 寄存器产生一定频率的发送脉冲。接收数据时钟可由 RDIVx 寄存器产生。在每个字或每帧数据传送的开始，串口要产生发送帧同步信号 TFSx 或接收帧同步信号 RFSx，它们同样也受寄存器 TDIVx 或 RDIVx 的控制。

与串口有关的寄存器属 IOP 寄存器，包括：STCTLx (串口发送控制寄存器，复位值 0h)、TXx(发送数据缓冲)、TDIVx(发送时钟及帧同步分频器)、MTCSx(多通道发送选择)、MTCCSx(多通道发送压扩选择)、SRCTLx(串口接收控制寄存器，复位值为 0h)、RXx(接收数据缓冲)、RDIVx(接收时钟及帧同步分频器)、MRCSx(多通道接收选择)、MRCCSx(多通道接收压扩选择)、SPATHx(串口途径长度，复位值为 01h)、KEYWDx(串口接收比较，仅应用于 ADSP21061)、KEYMASKx(串口接收比较屏蔽，仅应用于 ADSP21061)。

若接收缓冲满时，仍有新数据被接收，则旧数据会被覆盖并给出接收溢出状态标志 ROVF=1；处理器核对一个空的接收缓冲读或者向一个满的发送缓冲写都会使执行被挂起，要防止这种挂起有两个方法：在读写前先检查 STCTLx/SRCTLx 中的相应状态标志，或者使 SYSCON 寄存器的 BHD=1。

2.3.6 ADSP2106x DSP 的汇编指令

ADSP2106x 有 48 bit 的超长指令字(VLIW)，一条指令包含多个可选操作，各操作之间用逗号隔开，分号表示指令结束。全部指令可分成如下 4 大组(IF 条件也是可选项)：

(1) 计算并行数据存取指令：

IF 条件　计算，数据存取或一个地址寄存器修改；

(2) 程序流控制：

IF 条件　跳转/调用/返回/循环，计算，或带一次数据存取；

(3) 直接数据存取：

操作数包含在指令中或直接寻址；

(4) 其它类指令：

位修改，位测试，空闲操作，等待中断。

1．指令形式

上述(1)、(2)两组指令充分利用了 ADSP2106x 的片内多个功能单元的并行操作特点，体

现了 ADSP2106x 的超级哈佛结构的高效特点。表 2.85 中列出了 ADSP2106x 的所有指令形式，条件(condition)和循环终止码列在表 2.86 中，计算(compute)指令将在后文再介绍。

表 2.85 中的竖线表示从其内部选择一条指令，表中符号的含义见表后解释。

表 2.85　ADSP2106x 的指令形式

类别	指令形式
计算和数据存取类指令	1.　(compute),　$\left\|\begin{matrix}DM(Ia,Mb)=dreg1\\ dreg1=DM(Ia,Mb)\end{matrix}\right\|,\left\|\begin{matrix}PM(Ic,Md)=dreg2\\ dreg2=PM(Ic,Md)\end{matrix}\right\|;$
	2.　IF (condition) compute;
	3a.　IF (condition) (compute),　$\left\|\begin{matrix}DM(Ia,Mb)\\ PM(Ic,Md)\end{matrix}\right\|=ureg;$
	3b.　IF (condition) (compute),　$\left\|\begin{matrix}DM(Mb,Ia)\\ PM(Md,Ic)\end{matrix}\right\|=ureg;$
	3c.　IF (condition) (compute),　$ureg=\left\|\begin{matrix}DM(Ia,Mb)\\ PM(Ic,Md)\end{matrix}\right\|;$
	3d.　IF (condition) (compute),　$ureg=\left\|\begin{matrix}DM(Mb,Ia)\\ PM(Md,Ic)\end{matrix}\right\|;$
	4a.　IF (condition) (compute),　$\left\|\begin{matrix}DM(Ia,<data6>)\\ PM(Ic,<data6>)\end{matrix}\right\|=dreg;$
	4b.　IF (condition) (compute),　$\left\|\begin{matrix}DM(<data6>,Ia)\\ PM(<data6>,Ic)\end{matrix}\right\|=dreg;$
	4c.　IF (condition) (compute),　$dreg=\left\|\begin{matrix}DM(Ia,<data6>)\\ PM(Ic,<data6>)\end{matrix}\right\|;$
	4d.　IF (condition) (compute),　$dreg=\left\|\begin{matrix}DM(<data6>,Ia)\\ PM(<data6>,Ic)\end{matrix}\right\|;$
	5.　IF (condition) (compute),　ureg1=ureg2;
	6a.　IF (condition) shiftimm,　$(\left\|\begin{matrix}DM(Ia,Mb)\\ PM(Ic,Md)\end{matrix}\right\|=dreg);$
	6b.　IF (condition) shiftimm,　$(dreg=\left\|\begin{matrix}DM(Ia,Mb)\\ PM(Ic,Md)\end{matrix}\right\|);$
	7.　IF (condition) (compute),　$MODIFY\left\|\begin{matrix}(Ia,Mb)\\ (Ic,Md)\end{matrix}\right\|;$

续表(一)

类别	指令形式
程序控制指令	8. IF (condition) JUMP { <addr24> / (PC,<reladdr24>) } { (DB) / (LA) / (CI) / (DB,LA) / (DB,CI) } ; IF (condition) CALL { <addr24> / (PC,<reladdr24>) } (DB) ;
	9. IF (condition) JUMP { (Md,Ic) / (PC,<reladdr6>) } { (DB) / (LA) / (CI) / (DB,LA) / (DB,CI) } , { (compute) / (ELSE compute) } ; IF (condition) CALL { (Md,Ic) / (PC,<reladdr6>) } (DB), { (compute) / (ELSE compute) } ;
	10. IF condition JUMP { (Md,Ic) / (PC,<reladdr6>) } , ELSE { (compute),DM(Ia,Mb)=dreg / (compute),dreg=DM(Ia,Mb) } ;
	11. IF (condition) RTS { (DB0 / (LR) / (DB,LR) } , { (compute) / (ELSE compute) } ; IF (condition) RTI(DB), { (compute) / (ELSE compute) } ;
	12. LCNTR = { <data16> / ureg } , DO { <addr24> / (PC,<reladdr24>) } UNTIL LCE ;
	13. DO { <addr24> / (PC,<reladdr24>) } UNTIL termination ;
直接数据存取指令	14a. { DM(<addr32>) / PM(<addr24>) } =ureg ; 14b. ureg= { DM(<addr32>) / PM(<addr24>) } ;
	15a. { DM(<data32>,Ia) / PM(<data24>,Ic) } = ureg ; 15b. ureg = { DM(<data32>,Ia) / PM(<data24>,Ic) } ;
	16. { DM(Ia,Mb) / PM(Ic,Md) } =<data32> ;
	17. ureg=<data32>;

续表(二)

类别	指　令　形　式
其它类指令	18.　BIT SET / CLR / TGL / TST / XOR sreg< data32 > ;
	19a.　MODIFY (Ia, < data32 >) / (Ic, < data24 >) ; 19b.　BITREV (I0, < data32 >) / (I8, < data24 >) ;
	20.　PUSH / POP LOOP ,　PUSH / POP STS ,　PUSH / POP PCSTK ，FLUSH CACHE;
	21.　NOP;
	22.　IDLE; IDLE16;
	23.　CJUMP function / (PC, < reladdr24 >) (DB) ; RFRAME;

表中符号的含义如下：

compute：算术逻辑单元、乘法器、移位器操作。

shiftimm：移位器立即数操作。

condition：状态条件。

termination：循环终止条件。

ureg：通用寄存器：dreg, sreg, DAG1～2 寄存器，程序控制寄存器，定时器，RX2～1。

sreg：系统寄存器：MODE2～1, IRPTL, IMASK, IMASKP，ASTAT，STKY，USTAT2～1。

dreg：数据寄存器：R0～15 或 F0～15。

Ia：I7～0(DAG1)。

Mb：M7～0(DAG1)。

Ic：I15～8(DAG2)。

Md：M15～8(DAG2)。

<datan>：n 位立即数。

<addrn>：n 位立即地址。

<reladdrn>：n 位立即数 PC 相对地址。

(DB)：迟延跳转，即执行后续两个指令后才跳转、调用或返回。

(LA)：循环中止(循环及 PC 堆栈弹出)。

(LR)：循环计数验证。

(CI)：清除中断。

( )：括号里面的内容表示可选项。

调用返回(RTS)和中断服务返回(RTI)的区别在于 RTI 要多做两个操作：第一，如果 ASTAT 和 MODE1 寄存器已经入栈的话，则应使它们出栈，这适用于 IRQ0～2、定时中断和 VIRPT 中断；第二，将 IRPTL 和 IMASKP 中的相应位清 0。

条件和循环终止码列于表 2.86 中，这些码都来自于状态寄存器 ASTAT 以及 MODE1、Flag 输入、循环计数器的各标志位。

表 2.86　条件和循环终止码

编号	助 记 名 称	功　　能	为 真 条 件
0	EQ	ALU=0	AZ=1
1	LT	ALU<0	注解(1)
2	LE	ALU≤0	注解(2)
3	AC	ALU 进位	AC=1
4	AV	ALU 溢出	AV=1
5	MV	乘法器溢出	MV=1
6	MS	乘法器符号	MN=1
7	SV	移位器溢出	SV=1
8	SZ	移位器为 0	SZ=1
9	FLAG0_IN	Flag0 输入值为 1	FI0=1
10	FLAG1_IN	Flag1 输入值为 1	FI1=1
11	FLAG2_IN	Flag2 输入值为 1	FI2=1
12	FLAG3_IN	Flag3 输入值为 1	FI3=1
13	TF	位测试标志	BTF=1
14	BM	主处理器	—
15	LCE NOT LCE	循环计数完(DO UNTIL) 循环计数未完	CURLCNTR=1 CURLCNTR≠1
16	NE	ALU≠0	AZ=0
17	GE	ALU≥0	注解(3)
18	GT	ALU>0	注解(4)
19	NOT AC	ALU 无进位	AC=0
20	NOT AV	ALU 无溢出	AV=0
21	NOT MV	乘法器无溢出	MV=0
22	NOT MS	乘法器符号为 0	MN=0
23	NOT SV	移位器无溢出	SV=0
24	NOT SZ	移位器不为 0	SZ=0
25	NOT FLAG0_IN	Flag0 输入为 0	FI0=0
26	NOT FLAG1_IN	Flag1 输入为 0	FI1=0
27	NOT FLAG2_IN	Flag2 输入为 0	FI2=0
28	NOT FLAG3_IN	Flag3 输入为 0	FI3=0
29	NOT TF	位测试标志为 0	BTF=0
30	NBM	非主处理器	—
31	FOREVER TRUE	常假 Always False (DO UNTIL) 常真 Always True (IF)	常值 常值

注：(1) [$\overline{AF}$ and (AN xor (AV and $\overline{ALUSAT}$)) or (AF and AN and $\overline{AZ}$)]=1；

(2) [$\overline{AF}$ and (AN xor (AV and $\overline{ALUSAT}$)) or (AF and AN)] or AZ=1；

(3) [$\overline{AF}$ and (AN xor (AV and $\overline{ALUSAT}$)) or (AF and AN and $\overline{AZ}$)]=0；

(4) [$\overline{AF}$ and (AN xor (AV and $\overline{ALUSAT}$)) or (AF and AN)]or AZ=0。

2．寻址模式

ADSP21x6x 的寻址模式比较简单，只有两种(见表 2.85)：间接寻址和直接寻址。间接寻址有两种地址修改模式：

• 后修改地址操作(M 寄存器或立即数放在 I 寄存器的后面)，例如 DM(Ia,Mb)=dreg 表示先将 dreg 的内容写到 Ia 指定的 DM 区存储单元中，后更新 Ia，即 Ia+Mb→Ia。

• 预修改地址操作(M 寄存器或立即数放在 I 寄存器的前面)，例如 DM(<data6>，Ia)=dreg 表示将 dreg 的内容写到 Ia+<data6>指定的 DM 区存储单元中，数据访问前后 Ia 的内容保持不变。

3．计算类操作

计算类操作大致分为：ALU 的加、减运算；移位器操作；乘法器运算；并行运算，即双加/减，并行 ALU/乘法器运算。

ADSP2106x 的所有运算都依赖于寄存器 R0～R15、MRF 和 MRB，操作数主要是 R0～R15 寄存器，而不能是存储器中的数值。浮点运算和定点运算都用同样的寄存器 R0～R15，但在浮点表达式中写为 F0～F15。

计算类操作和数据存取操作的表达式都采用代数符号表示形式，直观易记，简洁紧凑。

1) ALU 运算

ALU 定点运算和浮点运算的指令有所不同，分别列于表 2.87 和表 2.88 中。ALU 所有运算都是基于 R0～R15 进行的，这 16 个 40 bit 寄存器作定点运算时只用高 32 bit，作浮点运算时为 40 bit 浮点数，其中高 32 bit 符合 IEEE 标准，低 8 bit 为扩展精度而增加的尾数。

表 2.87　ALU 定点运算及其对标志位的影响

表 达 式	功　　能
Rn=Rx+Ry	加法，影响标志位 AZ、AV、AN、AC、AOS
Rn=Rx- Ry	减法，影响标志位 AZ、AV、AN、AC、AOS
Rn=Rx+Ry+CI	带进位加，影响标志位 AZ、AV、AN、AC、AOS
Rn=Rx- Ry+CI- 1	带进位减并减 1，1 相当于借位，影响标志位 AZ、AV、AN、AC、AOS
Rn=(Rx+Ry)/2	平均，影响标志位 AZ、AN、AC
COMP(Rx, Ry)	比较，影响标志位 AZ、AN、CACC
Rn=Rx+CI	加进位，影响标志位 AZ、AV、AN、AC、AOS
Rn=Rx+CI- 1	加进位并减 1，1 相当于借位，影响标志位 AZ、AV、AN、AC、AOS
Rn=Rx+1	加 1，影响标志位 AZ、AV、AN、AC、AOS
Rn=Rx－1	减 1，影响标志位 AZ、AV、AN、AC、AOS
Rn=- Rx	求负值，影响标志位 AZ、AV、AN、AC、AOS
Rn=ABS Rx	求绝对值，影响标志位 AZ、AV、AS、AOS
Rn=PASS Rx	Rx 经 ALU 传送到 Rn(影响了标志位)，影响标志位 AZ、AN
Rn=Rx AND Ry	位与，影响标志位 AZ、AN
Rn=Rx OR Ry	位或，影响标志位 AZ、AN
Rn=Rx XOR Ry	位异或，影响标志位 AZ、AN
Rn=NOT Rx	位取反，影响标志位 AZ、AN
Rn=MIN(Rx, Ry)	取最小值，影响标志位 AZ、AN
Rn=MAX(Rx, Ry)	取最大值，影响标志位 AZ、AN
Rn=CLIP Rx BY Ry	当 |Rx|<|Ry| 时，Rn=Rx；否 { Rx>0 时，Rn=|Ry|；Rx<0 时，Rn=-|Ry| } 影响标志位 AZ、AN

表 2.88　ALU 浮点运算及其对标志位的影响

表 达 式	功　　能
Fn=Fx+Fy	加，影响标志位 AZ、AV、AN、AI、AUS、AVS、AIS
Fn=Fx－Fy	减，影响标志位 AZ、AV、AN、AI、AUS、AVS、AIS
Fn=ABS(Fx+Fy)	和的绝对值，影响标志位 AZ、AV、AI、AUS、AVS、AIS
Fn=ABS(Fx－Fy)	差的绝对值，影响标志位 AZ、AV、AI、AUS、AVS、AIS
Fn=(Fx+Fy)/2	求平均，影响标志位 AZ、AN、AI、AUS、AIS
COMP(Fx,Fy)	比较，影响标志位 AZ、AN、AI、CACC、AIS
Fn=－Fx	求负值，影响标志位 AZ、AV、AN、AI、AVS、AIS
Fn=ABS Fx	求绝对值，影响标志位 AZ、AV、AS、AI、AVS、AIS
Fn=PASS Fx	Fx 经 ALU 传送到 Fn，影响标志位 AZ、AN、AI、AIS
Fn=RND Fx	将 Fx 截取 32 bit IEEE 浮点数送给 Fn，截断方法取决于 MODE1 的相应位，影响标志位 AZ、AV、AN、AI、AVS、AIS
Fn=SCALB Fx BY Ry	将整数 Ry 加到 Fx 的指数上，并送给 Fn，影响标志位 AZ、AV、AN、AI、AUS、AVS、AIS
Rn=MANT Fx	把 Fx 的尾数作为无符号整数送给 Rn，影响标志位 AZ、AV、AS、AI、AVS、AIS
Rn=LOGB Fx	把 Fx 的指数作为补码整数送给 Rn，影响标志位 AZ、AV、AN、AI、AVS、AIS
Rn=FIX Fx BY Ry	把 Ry 加到 Fx 的指数上，然后取 Fx 整数部分送给 Rn，截断方法取决于 MODE1 的相应设置位，影响标志位 AZ、AV、AN、AI、AUS、AVS、AIS
Rn=FIX Fx	取 Fx 整数部分送给 Rn，截断方法取决于 MODE1 的相应设置位，影响标志位 AZ、AV、AN、AI、AUS、AVS、AIS
Rn=TRUNC Fx BY Ry	把 Ry 加到 Fx 的指数上，取 Fx 整数部分送给 Rn，影响标志位 AZ、AV、AN、AI、AUS、AVS、AIS
Rn=TRUNC Fx	取 Fx 整数部分送给 Rn，影响标志位 AZ、AV、AN、AI、AUS、AVS、AIS
Fn=FLOAT Rx BY Ry	把 Rx 转换为浮点数传给 Fn，给 Fn 的指数加上 Ry，影响标志位 AZ、AV、AN、AUS、AVS
Fn=FLOAT Rx	把 Rx 转换为浮点数传给 Fn，影响标志位 AZ、AN
Fn=RECIPS Fx	为求 1/ Fx 而产生 8 bit 的初步结果，影响标志位 AZ、AV、AN、AI、AUS、AVS、AIS
Fn=RSQRTS Fx	为求 $1/\sqrt{Fx}$ 而产生 4 bit 的初步结果，影响标志位 AZ、AV、AN、AI、AVS、AIS
Fn=Fx COPYSIGN Fy	取 Fx 的指数和尾数及 Fy 的符号位送给 Fn，影响标志位 AZ、AN、AI、AIS
Fn=MIN (Fx,Fy)	求最小值，影响标志位 AZ、AN、AI、AIS
Fn=MAX(Fx,Fy)	求最大值，影响标志位 AZ、AN、AI、AIS
Fn=CLIP Fx BY Fy	当 |Fx|<|Fy|时，Fn=Fx；否则 $\begin{cases} Fx>0\text{ 时，}Fn=\lvert Fy\rvert \\ Fx<0\text{ 时，}Fn=-\lvert Fy\rvert \end{cases}$ 影响标志位 AZ、AN、AI、AIS

2) 移位器操作

移位器为 32 bit，完成对寄存器 R0～R15 的位操作，如表 2.89 所示。

表 2.89　移位器操作及其对标志位的影响

指　令	SZ SV SS	功　能
Rn=LSHIFT Rx BY Ry	* * 0	把 Rx 逻辑移 Ry 位
Rn=LSHIFT Rx BY<data8>	* * 0	把 Rx 逻辑移<data8>位，data 8 是 8 bit 立即数
Rn=Rn OR LSHIFT Rx BY Ry	* * 0	把 Rx 逻辑移 Ry 位，得到的值与 Rn 按位或
Rn=Rn OR LSHIFT Rx BY<data8>	* * 0	把 Rx 逻辑移<data8>位，得到的值与 Rn 按位或
Rn=ASHIFT Rx BY Ry	* * 0	把 Rx 算术移 Ry 位
Rn=ASHIFT Rx BY<data8>	* * 0	把 Rx 算术移<data8>位，data8 是 8 bit 立即数
Rn=Rn OR ASHIFT Rx BY Ry	* * 0	把 Rx 算术移 Ry 位，得到的值与 Rn 按位或
Rn=Rn OR ASHIFT Rx BY<data8>	* * 0	把 Rx 算术移<data8>位，得到的值与 Rn 按位或
Rn=ROT Rx BY Ry	* 0 0	把 Rx 循环移 Ry 位
Rn=ROT Rx BY <data8>	* 0 0	把 Rx 循环移<data8>位
Rn=BCLR Rx BY Ry	* * 0	将 Rx 的第 Ry 位清 0
Rn=BCLR Rx BY <data8>	* * 0	将 Rx 的第<data8>位清 0
Rn=BSET Rx BY Ry	* * 0	将 Rx 的第 Ry 位置位
Rn=BSET Rx BY <data8>	* * 0	将 Rx 的第<data8>位置位
Rn=BTGL Rx BY Ry	* * 0	将 Rx 的第 Ry 位取反
Rn=BTGL Rx BY<data8>	* * 0	将 Rx 的第<data8>位取反
BTST Rx BY Ry	* * 0	测试 Rx 的第 Ry 位是否为 1
BTST Rx BY<data8>	* * 0	测试 Rx 的第<data8>位是否为 1
Rn=FDEP Rx BY Ry Rn=FDEP Rx BY<bit6>:<len6>	* * 0	在 Rn 从<bit6>位起始，长<len6>的位段上，放入 Rx 的从位 8 开始，同样长度的位段。 <bit6>和<len6>为立即数，或对应 Ry 的相应位段：<bit6>在 Ry 的 bit8～bit13；<len6>在 Ry 的 bit14～bit19(下同)
Rn=Rn OR FDEP Rx BY Ry Rn=Rn OR FDEP Rx BY<bit6>:<len6>	* * 0	在 Rn 从<bit6>位起始，长<len6>的位段上，放入 Rx 的从位 8 开始，同样长度的位段，放入前与 Rn 中相应位段按“位或”
Rn=FDEP Rx BY Ry (SE) Rn=FDEP Rx BY<bit6>:<len6> (SE)	* * 0	在 Rn 从<bit6>位起始，长<len6>的位段上，放入 Rx 从位 8 开始，同样长度的位段，放入前应对位段左边位作符号扩展
Rn=Rn OR FDEP Rx BY Ry (SE) Rn=Rn OR FDEP Rx BY<bit6>:<len6> (SE)	* * 0	在 Rn 从<bit6>位起始，长<len6>的位段上，放入 Rx 从位 0 开始，同样长度的位段，并作符号扩展，而且放置前与 Rn 按“位或”
Rn=FEXT Rx BY Ry Rn=FEXT Rx BY<bit6>:<len6>	* * 0	在 Rx 从<bit6>位起始，长<len6>的位段上作截取，结果放入 Rn 从位 8 开始，同样长度的位段上
Rn=FEXT Rx BY Ry (SE) Rn=FEXT Rx BY<bit6>:<len6> (SE)	* * 0	在 Rx 从<bit6>位起始，长<len6>的位段上作截取，结果放入 Rn 从位 8 开始，同样长度的位段上，并作符号扩展
Rn=EXP Rx	* 0 *	把 Rx 的指数部分放到 Rn 的 shf8 段，shf8 在 Rn 的 bit8～bit15(下同)
Rn=EXP Rx (EX)	* 0 *	把 Rx 的指数部分放到 Rn 的 shf8 段，但 AV 为 1 时，应加 1
Rn=LEFTZ Rx	* * 0	统计 Rx 左起连续“0”的个数，送到 Rn 中
Rn=LEFTO Rx	* * 0	统计 Rx 左起连续“1”的个数，送到 Rn 中
Rn=FPACK Fx	* 0 *	将 32 bit 浮点(Fx)转换成 16 bit 浮点(Rn)
Fn=FUNPACK Rx	0 0 0	将 16 bit 浮点(Rx)转换成 32 bit 浮点(Fn)

注：*表示其值取决于数据。

逻辑移位(LSHIFT)时 Rn 的低 8 bit 不参与，右移时高位填 0，低位送进位，左移时高位送进位，低位填 0；算术移位(ASHIFT)时 Rn 的低 8 bit 不参与，右移时高位符号扩展，低位送进位，左移时高位送进位，低位填 0。移位量为正代表左移，为负代表右移。

3) 乘法器操作

ADSP2106x 的乘法器的定点操作和浮点操作有显著差别，如表 2.90 所示。浮点数动态范围大，不用考虑溢出，采用 40 bit 浮点寄存器。定点乘法则采用了两个 80 bit 的乘法累加器 MRF、MRB 来保存两个 32 bit 定点数的乘积以改进动态范围，并且可以为定点乘法设置多种操作模式。

表 2.90　乘法指令及其对标志位的影响

指　令	ASTAT 标志位				STKY 标志位				功　能
	MU	MN	MV	MI	MUS	MOS	MVS	MIS	
Rn=Rx*Ry　mod1	*	*	*	0	—	**	—	—	对小数乘积结果取中 32 位 MR1，对整数乘积结果取低 32 位 MR0
MRF=Rx*Ry　mod1	*	*	*	0	—	**	—	—	80 位的结果全部保存在 MRF 中
MRB=Rx*Ry　mod1	*	*	*	0	—	**	—	—	80 位的结果全部保存在 MRB 中
Rn=MRF+Rx*Ry　mod1	*	*	*	0	—	**	—	—	Rx 与 Ry 相乘后再与 MRF 相加，最后放入 Rn 中，对小数取中 32 位 MR1，对整数取低 32 位 MR0
Rn=MRB+Rx*Ry　mod1	*	*	*	0	—	**	—	—	同上
MRF=MRF+Rx*Ry　mod1	*	*	*	0	—	**	—	—	Rx 与 Ry 相乘后再与 MRF 相加，80 位的结果全部保存在 MRF 中
MRB=MRB+Rx*Ry mod1	*	*	*	0	—	**	—	—	同上
Rn=MRF- Rx*Ry mod1	*	*	*	0	—	**	—	—	Rx 与 Ry 相乘后再与 MRF 相减，最后放入 Rn 中，对小数取中 32 位 MR1，对整数取低 32 位 MR0
Rn=MRB- Rx*Ry　mod1	*	*	*	0	—	**	—	—	同上
MRF=MRF- Rx*Ry　mod1	*	*	*	0	—	**	—	—	Rx 与 Ry 相乘后再与 MRF 相减，80 位的结果全部保存在 MRF 中
MRB=MRB- Rx*Ry　mod1	*	*	*	0	—	**	—	—	同上
Rn=SAT MRF mod2	*	*	*	0	—	**	—	—	对小数取中 32 位 MR1，对整数取低 32 位 MR0，如果 MRF 的值大于指定数据格式的最大值，MRF 应放入最大值，否则 MRF 不受影响
Rn=SAT MRB mod2	*	*	*	0	—	**	—	—	同上
MRF=SAT MRF mod2	*	*	*	0	—	**	—	—	80 位的结果全部保存在 MRF 中，如果 MRF 的值大于指定数据格式的最大值，MRF 应放入最大值，否则 MRF 不受影响

续表

指　令	ASTAT 标志位				STKY 标志位				功　能
	MU	MN	MV	MI	MUS	MOS	MVS	MIS	
MRB=SAT MRB mod2	*	*	*	0	—	**	—	—	同上
Rn=RND MRF mod2	*	*	*	0	—	**	—	—	对小数取中 32 位 MR1，对整数取低 32 位 MR0，但 MRF 应在 MR1、MR0 边界处取整
Rn=RND MRB mod2	*	*	*	0	—	**	—	—	同上
MRF=RND MRF mod2	*	*	*	0	—	**	—	—	80 位的结果全部保存在 MRF 中，但 MRF 应在 MR1、MR0 边界处取整
MRB=RND MRB mod2	*	*	*	0	—	**	—	—	同上
MRF=0	0	0	0	0	—	—	—	—	MRF 所有 80 位都清 0
MRB=0	0	0	0	0	—	—	—	—	MRB 所有 80 位都清 0
MRxF=Rn	0	0	0	0	—	—	—	—	把 Rn 高 32 位放入指定的 MRxF 中
MRxB=Rn	0	0	0	0	—	—	—	—	把 Rn 高 32 位放入指定的 MRxB 中
Rn=MRxF	0	0	0	0	—	—	—	—	把指定的 MRxF 放在 Rn 中，浮点扩展精度域清 0
Rn=MRxB	0	0	0	0	—	—	—	—	把指定的 MRxB 放在 Rn 中，浮点扩展精度域清 0
Fn=Fx*Fy	*	*	*	*	—	**	**	**	浮点乘法

注：(1) MRxF 代表 MR2F，MR1F，MR0F；MRxB 代表 MR2B，MR1B，MR0B。

(2) *表示置位或清 0，取决于结果；**表示可能置位或清 0，取决于结果。

对 ADSP2106x 来说，一个定点数既可以看成是整数也可当作纯小数，而对整数和小数乘积结果的处理方式是不同的。两个 32 bit 整数的乘积结果常从低 32 位取，而小数乘积结果则从高 32 位取值，显然后者不会溢出。MR 寄存器分成了高 16 bit(MR2)、中 32 bit(MR1)和低 32 bit(MR0)。小数相乘时，MR1 存放小数乘积结果，MR2 作为符号位，MR0 作为下溢出位；整数相乘时，MR2 和 MR1 都存放符号位(溢出位)，MR0 存放相乘结果。

R0～R15 与 MRF、MRB 传递数据时也比较特殊。当数据从 MR2 读出时，将高 16 bit 进行符号扩展成为 32 bit。将数据从 MR2、MR1、MR0 送到 Rn 时，Rn 的低 8 bit 填 0。同样，只有 Rn 的高 32 bit 才能写到 MR2、MR1、MR0 中。向 MR1 写数时，符号扩展到 MR2 的所有 16 bit，但向 MR0 写数时，MR2、MR1 都不会作符号扩展。

乘法器除了影响 ASTAT 中 4 个有关标志位外，还影响 STKY 寄存器的 4 个标志位。乘法器影响的标志位如下：ASTAT 寄存器中的 MU、MN、MV、MI 4 个标志位和 STKY 寄存器中的 MOS、MVS、MUS、MIS 4 个标志位。

定点乘法指令中 mod1 和 mod2 的选项如表 2.91 所示。

表 2.91　定点乘法 mod1 和 mod2 的选项

项　目	符　号　表　示
选项内容	S：输入量是有符号数 U：输入量是无符号数 I：输入量是整数 F：输入量是小数 FR：输入量是小数，结果写入寄存器时按最接近的值截取
Mod1	(SSI)、(SUI)、(USI)、(UUI)、(SSF)、(SUF)、(USF)、(UUF)、(SSFR)、(SUFR)、(USFR)、(UUFR)
Mod2	(SI)(只对 SAT 指令用)、(UI)(只对 SAT 指令用)、(SF)、(UF)

4) 并行指令

ADSP2106x 的各计算单元可同时进行并行运算，并行指令中间以逗号相隔。这些并行指令可分成以下三类：并行加、减；并行乘法、ALU 操作；并行乘法、加、减。其中并行乘、加、减运算使 ADSP2106x 完成 FFT 运算的速度大大提高。表 2.92 列出了各种并行指令形式。

表 2.92　多 运 算 指 令

指令类别	指　令　形　式
加、减	Ra=Rx+Ry, Rs=Rx－Ry Fa=Fx+Fy, Fs= Fx－Fy
乘、ALU 运算	Rm=R3～0*R7～4(SSFR) MRF=MRF+R3～0*R7～4(SSF) Rm=MRF+R3～0*R7～4(SSR) MRF=MRF- R3～0*R7～4(SSF) Rm=MRF- R3～0*R7～(SSFR) , Ra=R11～8+R15～12 Ra=R11～8- R15～12 Ra=(R11～8+R15～12)/2 Fm=F3～0*F7～4, Fa=F11～8+F15～12 Fa=F11～8- F15～12 Fa=FLOATR11～8 by R15～12 Ra=FIX F11～8 by R15～12 Fa=(F11～8+F15～12)/2 Fa=ABS F11～8 Fa=MAX(F11～8, F15～12) Fa=MIN(F11～8, F15～12)
乘、加、减	Rm=R3～0*R7～4(SSFR)，Ra=R11～8+R15～12，Rs=R11～8－R15～12 Fm=F3～0*R7～4，Fa=F11～8+F15～12，Fs=F11～8－F15～12

注：　Rm, Ra, Rs, Rx, Ry：任意寄存器(定点)。

Fm, Fa, Fs, Fx, Fy：任意寄存器(浮点)。

SSFR：x 输入为有符号数，y 输入为有符号数，且 x、y 都为小数，输出按最接近值截取。

SSF：x 输入为有符号数，y 输入为有符号数，且 x、y 都为小数。

4．其它类指令

(1) $\text{BIT}\left|\begin{array}{l}\text{SET}\\ \text{CLR}\\ \text{TGL}\\ \text{TST}\\ \text{XOR}\end{array}\right|\text{sreg}<\text{data32}>$

按照<data32>中置位的位(非 0 位)对系统寄存器 sreg 的相应位进行各种操作：置位(SET)、清 0(CLR)、取反(TGL)、测试(TST)、异或(XOR)。

(2) $\text{MODIFY}\left|\begin{array}{l}(\text{Ia},<\text{data32}>)\\ (\text{Ic},<\text{data24}>)\end{array}\right|$

修改 DAG1 的 I0～7 寄存器或 DAG2 的 I8～15 寄存器。

$\text{BITREV}\left|\begin{array}{l}(\text{I0},<\text{data32}>)\\ (\text{I8},<\text{data24}>)\end{array}\right|$

完成位反序方式的地址寄存器修改，地址寄存器只能为 I0(DAG1)和 I8(DAG2)。

(3) $\left|\begin{array}{c}\text{PUSH}\\ \text{POP}\end{array}\right|\text{LOOP},\left|\begin{array}{c}\text{PUSH}\\ \text{POP}\end{array}\right|\text{STS},\left|\begin{array}{c}\text{PUSH}\\ \text{POP}\end{array}\right|\text{PCSTK},\text{FLUSH CACHE}$

对下述对象作压栈或出栈操作：循环地址和循环计数器(LOOP)；状态寄存器(STS)；PC 堆栈(PCSTK)；刷新(清除)cache。这 4 个操作可以在一个周期内完成。

(4) NOP：空操作，仅程序地址增 1。

(5) IDLE：执行类似 NOP 的空操作，DSP 进入低功耗状态，PC 指针不变，直到某个中断出现后，才去响应中断，并执行后续指令。

(6) IDLE16：仅适用于 ADSP21061，类似 IDLE 指令，但 DSP 的时钟频率等于输入时钟的 1/16，其功耗更低。

(7) $\text{CJUMP}\left|\begin{array}{l}\text{函数名}\\ (\text{PC},<\text{reladdr24}>)\end{array}\right|(\text{DB})$

仅由 C 编译器产生，相当于通常跳转指令操作和帧指针 I6、堆栈指针 I7 的保存操作。

(8) RFRAME：仅由 C 编译器产生，相当于帧指针 I6、堆栈指针 I7 的恢复操作。

5．汇编程序编写

汇编程序中包含前面介绍的处理器指令和伪指令。伪指令是汇编器命令，用来控制汇编过程和定义数据结构等，表 2.93 中列出了 ADSP21xxx 汇编器的部分伪指令。

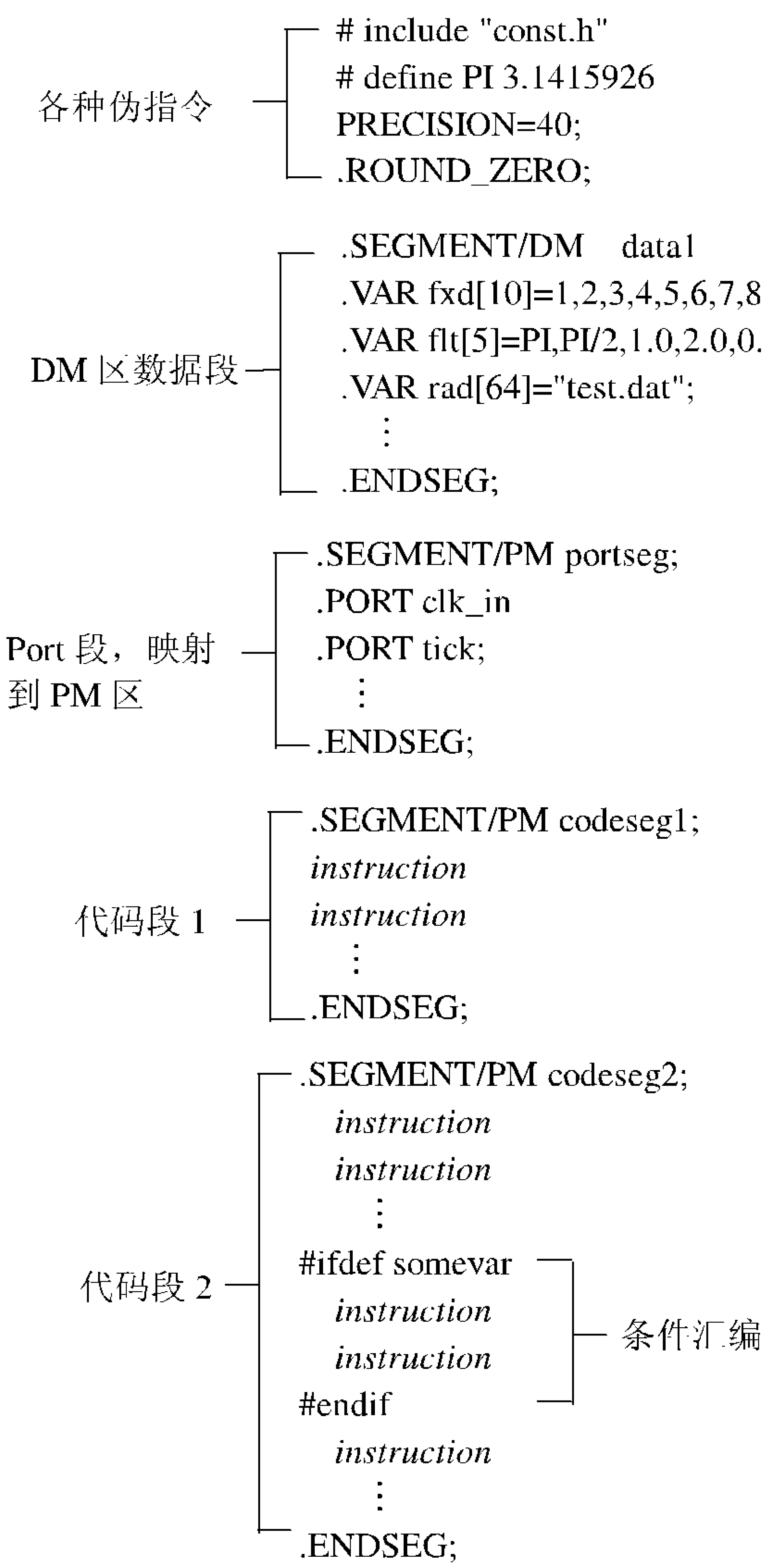
表 2.93　ADSP21xxx 汇编器的部分伪指令

伪　指　令	说　　明
.segment [/*TYPE sectionName sectionType*]	标识一段代码或数据段的起始
.endseg	标识一段代码或数据段的结束
.precision=32/40	指定浮点精度，默认为 32 bit
.round_*mode*	常数定义和变量初始化时的浮点近似方法，*mode* 为：nearest(向最接近的方向近似，默认)、zero(向零方向近似)、minus(负向近似)、plus(正向近似)

续表

伪　指　令	说　　明
.port　*portName*	定义一个存储器映射的 I/O 口，*portName* 为 I/O 端口名
.var *varName*1 [,*varName*2,]； .var *buffer*[ ]={*initExpression*1, *initExpression*2,}； .var *bufferName* [ ] ="*fileName* " ; .var bufferName[*length*]=*initExpression*1 ,*initExpression*2,;	定义数据变量或数据缓冲
.global *symbolName*1 [,*symbolName*2 ,…]	声明全局符号
.extern *symbolName*1 [,*symbolName*2 ,　…]	声明此汇编模块中用到的在其它模块中定义的全局符号(利用.global)
.newpage	在列表文件中开启新页
.align *expression*	把下面的代码或数据指定到 *expression* 地址边界上

下面给出了一段汇编程序编写格式的例子：



#include、#define、#ifdef 和#endif 等与 C/C++语言中的含义相同，都是 ADSP21xxx 汇编器的预处理伪指令。ADSP21xxx 汇编程序类似于 C/C++程序，而且其汇编指令本身也像 C/C++语言。汇编表达式也与 C/C++类似，其运算符含义如下(优先顺序自上而下)：

( )	括号
~　-	求补、负号
*　/　%　+　-	乘、除、求余、加、减
<<　>>	按位左移、右移
&	按位与
|	按位或
^	按位异或
@datbuff	表示求 datbuff 这个数据缓冲区的长度(以 32 bit 字计算)

利用 VisualDSP++集成环境开发时，还需要编写链接描述文件(*.ldf)，用来定义系统配置、存储器分配、需要链接的所有目标文件和目标库以及指示链接器如何链接等。链接描述文件(*.ldf)的定义在后文的示例中再介绍。

下面的代码为用汇编语言编写的 DFT 程序及其链接描述文件(针对 ADSP21062 DSP)：

```
#include "def21060.h"               /*包含 def21060.h 头文件*/
#define N 64                        /* 定义输入点数的长度 N*/

.SEGMENT/DM      dm_data;           /* 在 DM 存储区中定义变量 */
.VAR input[N]= "test64.dat";        /*在汇编程序中插入 dat 文件，用来初始化输入缓冲*/
.VAR real[N];
.VAR imag[N];
.ENDSEG;

.SEGMENT/PM      pm_data;           /* 在 DM 存储区中定义变量 */
.VAR sine[N]= "sin64.dat";          /* 旋转因子系数表*/
.ENDSEG;

.SEGMENT/PM      pm_rsti;           /*复位中断矢量，4 条指令*/
    NOP;
    USTAT2= 0x108421;               /* 1st instr. to be executed after reset */
    DM(WAIT)=USTAT2;                /* Set external memory waitstates to 0 */
    JUMP start;                     /* 跳到主程序的入口处*/
.ENDSEG;

.SEGMENT/PM    pm_code;             /* 设置开始调用 DFT 子程序*/
start:           M1=1;
    M9=1;
    B0=input;
    L0=@input;                      /* @input 表示求 input 输入缓冲的长度，循环缓冲 */
    I1=imag;
    L1=0;
    CALL dft (DB);                  /* 延迟调用 DFT 子程序，同时执行下面两条指令*/
```

```
            I2=real;
            L2=0;
    end:              IDLE;

    /*__________________________DFT 子程序体__________________________*/
dft:            B8=sine;              /* Sine pointer */
       L8=@sine;
            B9=sine;                  /* Derive cosine from sine by */
            I9=sine+N/4;              /* shifting pointer over 2pi/4 */
            L9=@sine;                 /* and using a circular buffer.*/
            I10=0;                    /* I10 is used to increment the */
            L10=0;                    /* frequency of sine lookup.*/
            F15=0;                    /* Zero to clear accumulators */
            LCNTR=N, DO outer UNTIL LCE;          /*外层循环*/
             F8=PASS F15, M8=I10;                 /* Update frequency */
             F9=PASS F15, F0=DM(I0,M1), F5=PM(I9, M8);
             F12=F0*F5, F4=PM(I8,M8);
             LCNTR=N-1, DO inner UNTIL LCE;       /*内层循环*/
              F13=F0*F4, F9=F9+F12, F0=DM(I0,M1), F5=PM(I9,M8);
inner:              F12=F0*F5, F8=F8-F13, F4=PM(I8,M8);
              F13=F0*F4, F9=F9+F12;
              F8=F8- F13,    DM(I2,M1)=F9;        /* Write real result */
              MODIFY(I10, M9);                    /* Increment frequency */
outer:            DM(I1, M1)=F8;                  /* Write imaginary result */
          RTS;
.ENDSEG;
```

def21060.h 是重要的头文件，包含系统和 IOP 寄存器的位与地址定义，其中包括以通用符号名称表示的宏定义。对于其它头文件，用户可以查阅相关文献，这里不再一一介绍。

下面以 ADSP21062 DSP 为例，给出上述 DFT 汇编程序对应的链接描述文件。

```
ARCHITECTURE(ADSP-21062)                  // 目标 DSP 为 ADSP21062
SEARCH_DIR( $ADI_DSP\21k\lib)             // 指定库函数搜索路径
$LIBRARIES = lib060.dlb ;                 // 链接库文件
$OBJECTS = $COMMAND_LINE_OBJECTS ;        // 链接目标文件，在 Visual DSP++命令行中输
                                          // 入，在第 4 章再介绍

MEMORY    // 定义目标 DSP 的物理存储器段
{
    pm_rsti { TYPE(PM RAM) START(0x00020004) END(0x00020009) WIDTH(48) }
 pm_code { TYPE(PM RAM) START(0x00020100) END(0x000207ff) WIDTH(48) }
    pm_data { TYPE(PM RAM) START(0x00023000) END(0x00027fff) WIDTH(32) }

    dm_data { TYPE(DM RAM) START(0x00028000) END(0x000287ff) WIDTH(32) }
} // End MEMORY

PROCESSOR p0    // 为 DSP p0 分配目标代码和数据段
```

```
{
   LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST) // Other object files to link against.
   OUTPUT( $COMMAND_LINE_OUTPUT_FILE )          // Resulting executable file name,
                                                // 在命令行中指定

   SECTIONS    // 把程序中的代码和数据段指定到 DSP 的物理存储段中
   {
       //  Map the sections specified in the program files to sections declared in
       //  MEMORY and use these sections to create the object file for processor p0.
   .pm_rsti
       {
         INPUT_SECTIONS( $OBJECTS(pm_rsti) $LIBRARIES(pm_rsti))
       } >pm_rsti
   .pm_code
       {
            INPUT_SECTIONS( $OBJECTS(pm_code) $LIBRARIES(pm_code))
       } >pm_code
   .pm_data
       {
           INPUT_SECTIONS( $OBJECTS(pm_data) $LIBRARIES(pm_data))
     } >pm_data
   .dm_data
       {
            INPUT_SECTIONS( $OBJECTS(dm_data) $LIBRARIES(dm_data))
       } >dm_data

  }// End SECTIONS
} // End p0
```

.dlb 为归档器输出的档案库文件(包含一个或多个目标文件)，链接器在档案文件中搜索代码中用到的模块。.doj 为汇编器输出的目标文件，包含可重新定位的代码段和调试信息。

VisualDSP++安装目录\21k\lib(对于 ADSP210xx)和\211k\lib(对于 ADSP211xx)提供了如下目标库，用户可以在连接描述文件中加入应用程序中需要调用的库。

060_cpp_hdr.doj:　C++程序的运行时(runtime)库文件，包含复位时初始化程序并调用 main( )主函数。

060 _hdr.doj：C 程序的运行时(runtime)库文件，同 060_cpp_hdr.doj。

lib060.dlb：运行时库函数。

libc.dlb：C 语言运行时库函数。

libcpp.dlb：C++语言运行时库函数。

libcpprt.dlb：C++语言运行时库函数。

libio32.dlb：I/O 库函数(32 位)。

libio64.dlb：I/O 库函数(64 位)。

VisualDSP++安装目录\211k\lib 中有与上述一一对应的库，如果目标 DSP 为 ADSP211xx 系列，则必须在链接描述文件中加入此目录中的库。

如果应用程序调用库中的模块，还必须在 C/C++或汇编程序中利用#include 预处理命令包含相应的头文件(*.h)，头文件包含调用函数的原型声明。关于头文件及其函数，用户可以查阅相关文献，这里不再详细介绍。

2.3.7　ADSP21x6x DSP 的 C/C++语言编程

C/C++编译器的操作方法在第 4 章集成开发环境 VisualDSP++中再作介绍，本节只对有关 ADSP21xxx 的 C/C++语言编程的注意问题进行说明。

1．C/C++编译器的输出段

C/C++程序经 ADSP21xxx 的 C/C++编译器编译后，自动产生以下代码和数据段：

seg_pmco：包含编译器生成的所有程序指令，此段必须放入 PM 存储区。

seg_dmda：包含默认的全局和静态变量以及字符串常数，此段必须放入 DM 存储区。

seg_pmda：包含 PM 区数据变量，此段必须放入 PM 存储区。

seg_stak：编译器的系统堆栈空间，用于保存局部变量、函数返回地址和参数传递等，默认的堆栈大小为 4 K 32 bit 存储空间。ADSP21xxx 没有专门的堆栈指针，利用 I6(帧指针)和 I7(堆栈指针)来管理堆栈。堆栈段必须放入 DM 存储区。

seg_heap：为动态存储空间分配保留空间，C/C++程序利用 malloc、calloc、realloc 函数来动态分配存储空间，此段必须放入 DM 存储区。当然，如果 C/C++程序中没有动态分配存储空间，此段就不会产生。

seg_init：包含系统的初始化数据，用来初始化全局和静态变量等，此段必须放入 PM 存储区。

seg_rth：包含系统初始化代码和中断矢量表，此段必须放入 PM 的中断矢量表存储区中。

用户也可以在 C/C++程序中利用 segment 命令来定义自己的代码或数据段，例如：

```
segment("ext_data")  int  temp;
```

C/C++编译器会把 temp 变量放入到 ext_data 段中，否则 C/C++程序中的所有代码和数据分别放入上述默认段中。在汇编程序中，用户利用.segment 和.endseg 伪指令来定义代码和数据段。例如：

```
.segment/DM  dm_data
.var fxd[10]=1,2,3,4,5,6,7,8,9,10;
.var flt[5]=3.14,3.14/2,1.0,2.0,2.0/3.0;
.var rad[64]='test.dat';
.endseg
.segment /PM  pm_code          /*定义 asmcode 代码段*/
    r0 = 0x1234;               /*smcode 段中的代码*/
    r1 = 0x4567;
    r2 = r1 + r2;
.endseg
```

上例中，用户在汇编程序中定义了一个数据段 dm_data(放在 DM 区)和一个代码段 pm_code(放在 PM 段)。

C/C++程序和汇编程序链接时，用户必须利用结构文件(*.ach)或链接描述文件(*.ldf)把以上 C/C++编译器产生的代码和数据段以及汇编程序中定义的数据和代码段分配到相应的

存储区中。

下面给出了一个包含 C/C++程序和汇编程序的链接描述文件的例子(adsp21062)。

```
ARCHITECTURE(ADSP-21062)            //指定 DSP 类型
SEARCH_DIR( $ADI_DSP\21k\lib )      //指定链接器的搜索路径，路径名之间以分号相隔
// The lib060.dlb must come before libc.dlb because libc.dlb has some 21020
// specific code and data
$LIBRARIES = lib060.dlb, libc.dlb, libio32.dlb;
//定义库列表宏，对下文中出现的$LIBRARIES，链接器自动用此库列表替代。
// Libraries from the command line are included in COMMAND_LINE_OBJECTS.
$OBJECTS = 060_hdr.doj, $COMMAND_LINE_OBJECTS;
//定义目标文件列表宏，对下文中出现的 $OBJECTS，链接器自动用此目标文件列表替代。

MEMORY   //定义目标系统的物理存储器段，用以在 SECTIONS{}命令中把程序中的代码和数据
         //段指定到这些物理存储器段中
{
  seg_rth   { TYPE(PM RAM) START(0x00020000) END(0x000200ff) WIDTH(48) }
//C/C++编译器自动产生的代码段
  seg_init { TYPE(PM RAM) START(0x00020100) END(0x0002010f) WIDTH(48) }
  seg_pmco { TYPE(PM RAM) START(0x00020110) END(0x000240ff) WIDTH(48) }
  pm_code {TYPE(PM RAM) START(0x24100) END(0x000249ff) WIDTH(48)}
      //用户汇编程序的代码段
  seg_pmda { TYPE(PM RAM) START(0x00026a00) END(0x00027fff) WIDTH(32) }
  seg_dmda { TYPE(DM RAM) START(0x00028000) END(0x00029fff) WIDTH(32) }
  dm_data { TYPE(DM RAM) START(0x0002a000) END(0x0002bfff) WIDTH(32) }
      //用户汇编程序的数据段
  seg_heap { TYPE(DM RAM) START(0x0002e000) END(0x0002efff) WIDTH(32) }
  seg_stak { TYPE(DM RAM) START(0x0002f000) END(0x0002ffff) WIDTH(32) }
}
PROCESSOR p0   //DSP p0 的程序代码和数据段分配
{
  LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST)  //指示链接器在指定的可执行文件
//中搜索局部范围内不可识别的变量或符号，$COMMAND_LINE_LINK_AGAINST 表示从链接器的
//命令行中接收输入可执行文件
  OUTPUT( $COMMAND_LINE_OUTPUT_FILE )  //指定输出可执行代码名
// $COMMAND_LINE_OUTPUT_FILE 表示由链接器的 - O 开关选项来得到输出可执行代码名，在
// 第 4 章中再介绍
  SECTIONS    //把程序中的代码和数据段分配到 MEMORY{}中定义的物理存储器段中
  {
     c_rth     //用户在 SECTIONS 命令中定义的段名，可以为任意名
      {
          INPUT_SECTIONS( $OBJECTS(seg_rth) $LIBRARIES(seg_rth))
          // seg_rth 为 C 编译器输出的初始化代码和中断矢量表
      } >seg_rth     // seg_rth 为 MEMORY{}中定义的物理存储器段，物理存储器段名与程序中
                     //的段名不一定要求相同
     c_init          //C 程序输出的初始化段
      {
          INPUT_SECTIONS( $OBJECTS(seg_init) $LIBRARIES(seg_init))
      } >seg_init
```

```
        c_pmco          //C 程序输出的代码段
         {
              INPUT_SECTIONS( $OBJECTS(seg_pmco) $LIBRARIES(seg_pmco))
          } >seg_pmco
      asm_code       //汇编程序的代码段
         {
              INPUT_SECTIONS( $OBJECTS(pm_code) $LIBRARIES(pm_code))
          } > pm_code
        c_pmda      //C 程序的 PM 数据段
          {
              INPUT_SECTIONS( $OBJECTS(seg_pmda) $LIBRARIES(seg_pmda))
          } >seg_pmda
        c_dmda      //C 程序的 DM 数据段
          {
              INPUT_SECTIONS( $OBJECTS(seg_dmda) $LIBRARIES(seg_dmda))
           } > seg_dmda
       asm_data      //汇编程序的 DM 数据段
          {
              INPUT_SECTIONS( $OBJECTS(dm_data) $LIBRARIES(dm_data))
           } > dm_data
        c_stack   //为应用程序分配系统堆栈
          {
              ldf_stack_space = .;   //当前位置指针
              ldf_stack_length = MEMORY_SIZEOF(seg_stak);
           } > seg_stak
     c_heap   //为应用程序分配动态存储空间
          {
             ldf_heap_space = .;   //当前位置指针
               ldf_heap_end = ldf_heap_space + MEMORY_SIZEOF(seg_heap) - 1;
               ldf_heap_length = ldf_heap_end- ldf_heap_space;
          } > seg_heap
     }     // 结束 SECTIONS 命令范围
  }        // 结束 PROCESSOR p0 命令范围
```

2．数据类型

ADSP21xxx 的 C/C++语言中定义的数据类型如表 2.94 所示，表中列出了每一种数据类型的位长、表示方法和取值范围。

表 2.94　ADSP21xxx 的 C/C++语言中定义的数据类型

数据类型	位数	表示方法	数值范围
char,int,short, short int, long int, long	32	补码	–2 147 483 648～2 147 483 647
unsigned int, unsigned short,unsigned long int, unsigned char	32	原码	0～4 294 967 295
float	32	IEEE 32 bit 浮点格式	1.175494e–38～3.40282346e+38
double	32 或 64	IEEE 32 bit 浮点格式 或 IEEE 64 bit 浮点格式	如果利用 - double - size - 64 命令行开关选项进行编译，则为 64 bit 双精度浮点格式
long double	64	IEEE 64 bit 浮点格式	2.22507385e- 308～1.79769313e+308
pointer	32	原码	0～4 294 967 295
fract	32	32 bit 定点	- 1～+1

在为变量或常数分配存储空间时，一定要特别注意这些数据类型的位数及其数值范围，否则可能会导致错误结果。

3. 在 C/C++程序中调用汇编函数

关于 C/C++程序和汇编程序接口所遵守的原则，请参阅 2.1.7 节。下面针对 ADSP21xxx 编译器，介绍如何满足这些原则。

1) 函数名要求

在 C/C++主程序的开头首先把要调用的汇编函数声明为外部函数(例如：extern void sub();)，在汇编程序中此函数名前必须有一下画线(例如：_sub)，并用.global 伪指令声明为全局符号(例如：.global _sub)。对于不被 C 程序访问的其它符号前不要加下画线。

2) 参数传递

C/C++程序在调用汇编函数时，其输入参数的前三个 32 bit 数据类型分别放入 R4、R8、R12 中，其余的输入参数按逆序放入堆栈中，即最右边的输入参数先入栈(最高地址)，第四个输入参数最后入栈(最低地址)；如果输入参数的数据类型超过 32 bit，则此输入参数及其后所有输入参数也都放入堆栈中(无论是否为前三个输入参数)，次序为低 32 位先入栈(高地址)，高 32 位后入栈(低地址)。例如：

```
jolt(int a, float b, char c, float d);
```

前三个输入参数 a、b 和 c 分别传递到寄存器 R4、R8 和 R12 中，第四个输入参数 d 放入堆栈中。又例如：

```
cola(int w, long double x, char y, float z);
```

第一个输入参数 w 传递到寄存器 R4 中，因为第二个输入参数 x 为双精度数据类型，因此 x 放入堆栈中，其后的 y 和 z 输入参数也都放入堆栈中。

下面给出一个例子，演示汇编函数如何访问堆栈中的输入参数。

```
tab(int a, char b, float c, int d, int e, long double f);
```

前三个输入参数 a、b 和 c 分别传递到寄存器 R4、R8 和 R12 中，其余输入参数 d、e、f 都放入堆栈中。在汇编函数中，可以通过帧指针 I6 来访问输入参数：

```
r3=dm(1,i6);   /* 访问输入参数 d */
r3=dm(2,i6);   /* 访问输入参数 e */
r3=dm(3,i6);   /* 访问输入参数 f 的高 32 位 */
r3=dm(4,i6);   /* 访问输入参数 f 的低 32 位 */
```

如果汇编函数返回一个整数、字符、单精度浮点数或指针等 32 位数据类型，结果放入 R0 寄存器中；如果返回一个双精度浮点等双字数据类型，高 32 位放入 R0 中，低 32 位放入 R1 中；如果被调汇编函数返回一个数据阵，此数据阵的首地址放入 R1 中。

I7 寄存器用作堆栈指针(SP)，I6 寄存器用作帧指针(FP)。图 2.14 的例子示出了 C/C++程序开始调用汇编函数时完成的堆栈操作，假定汇编函数原型为：

```
int   asmsub(int a,int b,int c,int d,int e,int f);
```

汇编函数应该利用下面的代码段来返回并恢复堆栈指针和帧指针：

```
i12=dm(- 1, i6);          /* 从堆栈中取出返回地址(实际上，为前一个地址) */
jump(m14, i12)(DB);       /* 跳转(延迟)到返回地址处，m14 的值为 1 */
i7=i6;                    /* 在跳转指令的延迟内恢复 i7*/
i6=dm(0, I6);             /* 在跳转指令的延迟内恢复 i6*/
```

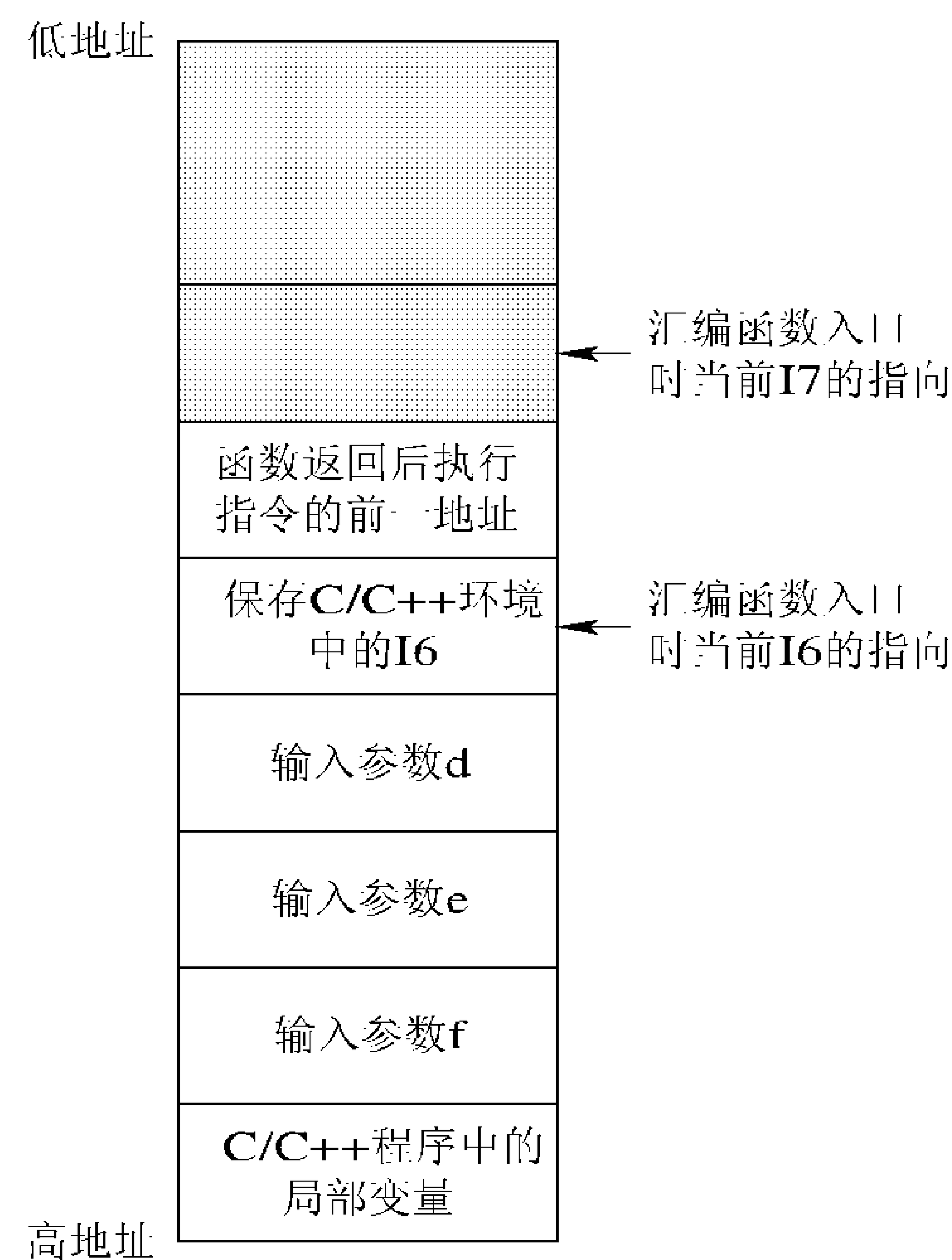


图 2.14　C/C++程序开始调用汇编函数时完成的堆栈操作

用户也可以利用 asm_sprt.h 头文件中定义的 exit 宏，来完成上述 4 条指令，asm_sprt.h 头文件中还定义了其它一些宏，用来简化一些常用的指令块，例如：entry、ccall(x)、reads(x)、puts(x)和 get(x)等宏。用户可以在 VisualDSP++的编辑窗中打开 asm_sprt.h，来查看这些宏定义。本书不再一一介绍。

3) 寄存器保护

经 C/C++ 编译器输出的汇编程序中会用到一些寄存器，而且编译器会对其中的某些寄存器有特殊要求。如果被调用的汇编函数中修改了这些寄存器，就必须对这些寄存器进行保护，否则会破坏 C/C++ 环境，从而导致错误结果。下面列出编译器用到的寄存器。

编译器用到的寄存器	值
m5,m13	固定为：0
m6,m14	固定为：1
m7,m15	固定为：-1
b6,b7	固定为：堆栈基址
l6,l7	固定为：堆栈长度
l0～l15(不包括 l6,l7)	固定为：0
i0,i1,i2,i3,i5,i8,i9,i10,i11,i14,i15	编译器修改
i6,i7	用于操作堆栈
m0,m1,m2,m3,m8.m9,m10,m11	编译器修改
r3,r5,r6,r7,r9,r10,r11,r13,r14,r15	编译器修改
mrf,mrb	编译器修改
mode1,mode2,ustat1,ustat2	编译器修改

下面的寄存器在被调用的汇编函数中可以任意修改，而无需保护。

r0, r1, r2, r4, r8, r12　　　其中 r0, r1, r4, r8, r12 用于参数传递

m4, m12

i4, i12, i13

4) 存储区分配及链接描述文件的修改

这部分内容在前面已作了介绍，这里不再重述。

下面给出一个在 C 程序中调用汇编函数的例子。在此例子中 C 程序调用汇编函数，而且汇编函数也访问 C 程序中定义的全局变量。

C 语言程序

```
#include <stdio.h>
void add5 (int a, int b, int c, int d, int e);
int sum_of_5;
main()
 {
  int a=1;
  int b=2;
  int c=3;
  int d=4;
  int e=5;
  add5(a,b,c,d,e);
 printf("sumresult=%d\n", sum_of_5);
 exit(0);
 }
```

汇编程序

```
#include <asm_sprt.h>
.section/pm   pm_code;
.extern   _sum_of_5;   /* C 程序中定义的全局变量 */
.global _add5;
_add5:  /*对 5 个输入参数求和输出 */
        /* 前三个输入参数分别放入 r4, r8, r12 中 */
   r4 = r4 + r8;    /*加第一个和第二个输入参数*/
   r4 = r4 + r12;   /*加第三个输入参数*/
/*下面利用 asm_sprt.h 中定义的 reads() 宏指令从堆栈中取
剩下的输入参数 */
   r8 = reads(1);   /*从堆栈中取第四个输入参数*/
   r4 = r4 + r8;    /*加第四个输入参数*/
   r8 = reads(2);   /*从堆栈中取第五个输入参数*/
   r4 = r4 + r8;    /*加第五个输入参数*/
   dm(_sum_of_5) = r4;  /*把结果放入全局变量_sum_of_5
中*/
   exit;  /*利用 exit 宏指令完成返回,相当于 i12=dm(-1,i6);
jump(m14,i12)(DB); i7=i6; i6=dm(0,I6); 四条指令*/
.endseg;
```

5) 编写支持 ADSP2116x 的 SIMD 模式的汇编函数

ADSP2116x 具有双运算核结构(PEx 和 PEy)，置位 MODE1 寄存器的 PEYEN 位，就启动了 SIMD 模式(即 PEy 运算核使能)。汇编指令在 SIMD 模式下的详细操作在 2.4.6 节中再详细介绍。

C/C++编译器不支持 SIMD 模式调用，因此可以在汇编函数的入口处设置为 SIMD 模式(PEYEN=1)，而在出口处变为调用前的 SISD 模式(PEYEN=0)。

为避免堆栈操作出错，对于堆栈操作最好在 SISD 模式下进行。如果 DSP 在 SIMD 模式下的执行过程中发生了中断，中断服务程序应对 PEYEN 位进行保护。

4．在 C/C++程序中插入汇编行

有时希望直接对 DSP 硬件操作，例如对寄存器操作或设置中断等，而 C/C++语言做不到这一点，可以通过在 C/C++程序中插入一条(多条)汇编指令来实现。在 C/C++程序中插入汇编指令的格式如下：

```
asm("汇编指令");     /*在 C/C++程序的当前位置插入汇编指令*/
```

例如：

```
asm("bit   set   mode2   0x40;");
```

表示插入汇编指令：bit set mode2 0x40;。

在 C/C++程序中插入汇编指令时，一定要特别注意不要破坏 C/C++程序的环境，例如对某一寄存器修改时，就可能会导致错误的结果。

5．C/C++程序和汇编程序中的全局变量互访

从汇编程序访问 C/C++程序中的全局或静态变量的方法很简单，只要在汇编程序中把此变量声明为外部符号(.extern _var；)就可以直接访问了，但要注意汇编程序中对应的变量名前应加一下划线。

类似地，从 C/C++ 程序访问汇编程序中的变量的方法也很简单：在汇编程序中将此变量声明为全局变量(.global _var；)，在 C/C++程序中将此变量声明为外部变量(extern int var;)后就可以直接访问了。

2.4 ADSP2116x DSP 的内部功能结构及源代码开发

2.4.1 ADSP2116x DSP 的功能和结构特点

AD 公司在 1999 年推出的这种高性能的 32 位浮点 ADSP2116x DSP 是在 ADSP2106x 的基础上开发的。它是对 ADSP2106x 的完善和提高，保持着与 ADSP2106x 代码的高度兼容，同时又具有高主频和 SIMD(Single－Instruction－Multiple－Data，单指令多数据流)的内部结构，因此它具有比 ADSP2106x 更强的运算能力。表 2.95 列出了 ADSP2116x 系列 DSP 的基本性能配置。例如，一片 100 MHz 主频的 ADSP21161 具有每秒 6 亿次的浮点运算速度，其处理能力相当于 5 片 40 MHz 主频的 ADSP2106x 的处理能力。

表 2. 95 ADSP2116x 系列 DSP 的基本配置

型　号	峰值处理速度	片内 RAM	链路口	串　口	时钟频率/MHz
ADSP21160N	570 MFLOPS	4 Mb	6	2	95
ADSP21160M	480 MFLOPS	4 Mb	6	2	80
ADSP21161	600 MFLOPS	1 Mb	2	4	100

ADSP2116x 保持了 ADSP2106x 大部分的结构，同时它又将 ADSP2106x 的性能和功能进一步扩展，主要表现在以下几个方面(以 ADSP21160 为例)。

1) 处理器核

ADSP21160 的计算带宽远大于 ADSP2106x，主要是由于提高了 CPU 频率和增加了另一套运算核：ALU、移位器、乘法器和寄存器组。这套新增加的运算核使得 DSP 可以并行处理多数据流(在 SIMD 模式下)。

程序控制器也不同于 ADSP2106x，主要有以下几个改进：新的中断矢量表定义，SIMD 堆栈和条件执行模式、新指令的解码。增加的新中断矢量可以发现非法的内存读取和支持新增加的 DMA 通道。链路口的中断控制移到一个新寄存器中。增加了新的模式堆栈和模式屏蔽，以改善切换时间。

数据地址产生器不同于 ADSP2106x，DAG2(PM 总线)也变成了 32 位(ADSP2106x 为 24 位)。DAG 支持新的存储器组织和长字传输能力。循环缓冲器能被中断迅速关闭，并能在中断结束时迅速恢复。数据可以从存储器向两运算核中的寄存器组进行广播。

2) 处理器内部总线

ADSP21160 的 PM、DM 和 I/O 数据总线都达到了 64 位，通过复路器和控制逻辑，可以在寄存器与存储器单元间进行 16/32/64 位数据传输。这样就可以通过广播把存储器单元的内容并行传送到两运算核的寄存器中，并允许两运算核寄存器中的内容在单周期内进行交换。

3) 存储器的组织管理

ADSP21160 的存储器组织支持每周期双字传输，这必须与支持 SIMD 的控制寄存器配合使用。

4) 外部口

数据总线增加到 64 位，增加了一个新的同步接口以提高本地总线速度，总线有分组能力。

• 主机接口。数据总线增加到 64 位。ADSP21160 不但支持 ADSP2106x 的异步主机接口协议，同时它还支持新的同步接口协议以实现最大吞吐量。ADSP21160 的主机/局部总线解死锁功能扩展到了 DMA 控制器。这种功能使得主机(或桥路)逻辑能够迫使局部总线避让，以允许主机首先完成它的操作。

• 多处理器接口。ADSP21160 的多处理器接口支持更大的数据吞吐量：总线宽度增加到 64 位；新的共享总线传输协议中的同步接口协议使得共享总线的周期时间改善。

5) DMA 控制器

DMA 通道增加到 14 个，比 ADSP2106x 增加了 4 个，不像 ADSP2106x 那样两个设备共享一个 DMA 通道，而是每个外设都有单独的 DMA 通道。新的打包模式支持 64 位外部／内部总线。为了解决死锁问题，当主机发出 $\overline{\text{HBR}}$ 和 $\overline{\text{SBTS}}$ 时，ADSP21160 的 DMA 控制器像主处理核那样放弃局部总线。

6) 链路口

每个链路口并行 8 bit 数据，具有高达核时钟的频率，而且链路口时钟频率可变。

7) 串口

提高了串口的最大时钟频率。

8) 指令系统

汇编源代码与 ADSP2106x 兼容，同时定义了一些新指令以支持新的功能。

图 2.15 为 ADSP21160 的内部结构框图。

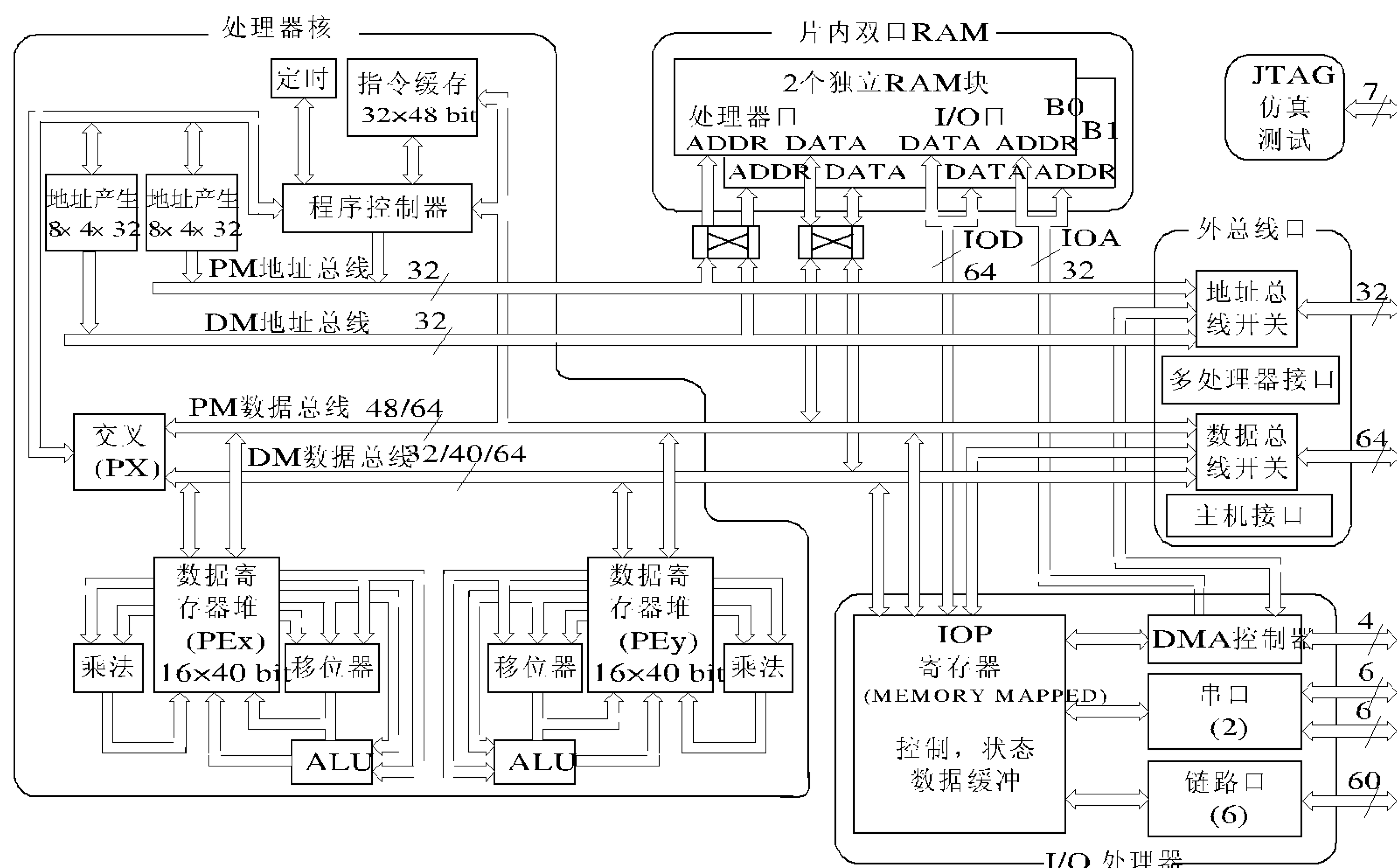


图 2.15　ADSP21160 的内部结构框图

2.4.2　CPU 核

1．运算核

ADSP21160 的 CPU 核包含了两个运算核：PEx 和 PEy。每个运算核包含一组数据寄存器和三个独立的计算单元：一个算术逻辑单元(ALU)、一个带定点累加器的乘法器和一个移位器。运算核兼容了早期的 ADSP21020 的运算核特点，具备高速、多功能的优点。

每个运算核都有 16 个数据寄存器，这些寄存器都具有另一套备份寄存器，可支持两者之间的高速切换。这此寄存器结合 CPU 核的超级哈佛结构，使得数据流在运算单元和内部存储器之间高速传输。运算核的这 16 个寄存器都是 40 位的，可用于定点和浮点运算。用于定点运算时，只做其中高 32 位的运算。用于浮点运算时可以选择 32 位或 40 位。当选择 40 位时，浮点运算单元读入 40 位数据，并将 40 位的结果送往 40 位宽的寄存器；当选择 32 位时，浮点运算单元接收 32 位的输入(低 8 位置 0)，结果也只取高 32 位，这与 IEEE 标准的 32 位浮点格式是一致的。80 位的 MR 寄存器(MRF 和 MRB)用于定点乘法结果的存放和累加。

运算核的操作模式受 MODE1 寄存器中位的控制。置位 MODE1 寄存器的 PEYEN 位，就会启动 PEy 运算核。

计算单元处理的数据有四种格式：

• 32 位定点数据占据 40 位数据寄存器的高 32 位。有两种定点数据格式，即整型和小数型。每种格式都可以是符号或无符号数据。所有的计算单元都从 40 位数据寄存器的高 32 位读入数据(忽略低 8 位)，并把计算结果放在数据寄存器的高 32 位(低 8 位填 0)。

• 32 位浮点，其格式符合 IEEE754/854 标准。

- 32 位浮点格式可以增添 8 个尾数位构成 40 位的扩展精度格式。
- 16 位短浮点格式，其数据格式为：1 个符号位、4 个指数位、11 个尾数位。它可以与 32 位浮点格式相互转换。

ADSP21160 的每个运算核都会对浮点运算结果出错提供额外的标志信息：乘法器和 ALU 在浮点运算过程中，对浮点的上溢、下溢和无效操作数都会置位算术状态寄存器(ASTATx 和 ASTATy)和辅助状态寄存器(STKYx 和 STKYy)的相应标志位，而且也会产生可屏蔽中断。因此程序中如果需要对浮点出错进行实时处理的话，有三种方法可供选择：

- 允许出错中断，通过中断服务程序来处理浮点出错信息。
- 通过条件指令测试 ASTATx 和 ASTATy 的标志位，对出错进行处理。
- 通过位测试指令测试 STKYx 和 STKYy 的标志位，对出错作出处理。

ALU 执行浮点或定点数据的算术操作和定点数据的逻辑操作。

乘法器执行浮点或定点的乘法，或定点的乘/加或乘/减运算。80 位的定点乘法结果寄存器(MR)比较特殊，它从高到低分为 3 个寄存器：MR2(高 16 位)、MR1(中 32 位)、MR0(低 32 位)，并与定点乘法器相连。如果乘法器的两个输入操作数都是小数型定点符号数，乘法器自动将结果左移 1 位以去掉多余的符号位，相乘结果放在 MR1 中(中 32 位)。如果乘法器输入为整型定点数，结果放在 MR0 中(低 32 位)。当从 MR2 中读取 32 位数据时，高 16 位符号扩展。当从 MR1、MR0 向 40 位寄存器送数时，低 8 位填 0。反之向 MR1、MR0 写数时，40 位寄存器的高 32 位被写入。如果是写入到 MR1，MR2 则作符号扩展，但写入到 MR0 时，MR2 不作符号扩展。ADSP21160 有两个 MR 寄存器：MRF、MRB，虽然可称为前台(MRF)和后台(MRB)，但任何时刻都可以访问任意一个。

移位器执行逻辑和算术移位、位操作、位段抽取和放置、取指数操作等。

所有的计算操作都是在单周期内完成的，所有的这些计算单元都是并行连接的，任何计算单元的输出在下一指令周期可充当任何计算单元的输入(即不像 TMS320C5000 和 C6000 那样，存在指令延迟间隙)。在多运算指令中，ALU 和乘法器同时并行工作。

运算核 PEx 无论是在 SISD，还是在 SIMD 模式下都参与运算。

运算核 PEy 只在 SIMD 模式下与 PEx 同步处理指令，而在 SISD 模式下不参与运算，即处于空闲状态。

2．程序控制器

程序控制器执行程序流控制，提供 DSP 要执行的下一条指令的地址。然而要执行的指令地址并不一定是顺序的，如程序中经常会用到循环(LOOP)、函数调用(CALL)、跳转(JUMP)、中断服务、等待(IDLE)等，程序控制器必须判断出下一条要执行的指令地址。

当取指地址计算出来后，就进入指令流水线中(由取指地址寄存器 FADDR、译码地址寄存器 DADDR、程序计数器 PC 组成)。ADSP21160 采用流水线方式执行每一条指令，每条指令包括取指、译码、执行三个周期。为消除跳转指令对流水线的影响，ADSP21160 支持带两级延迟的跳转/调用/返回指令。

程序控制器和循环堆栈指针相结合可以支持最多 6 级的无开销嵌套循环，每层循环都可以单周期退出。

指令 cache 使得 CPU 在单周期内完成从存储器 PM 区中读取数据和从指令 cache 中取指令的操作。

ADSP21160 对指令缓存(cache)的管理方式与其它 DSP 不同，它不是对要读取的每一条指令都进行 cache 地址比较，只有当 PM、DM 总线都用于存取数据时，才进行 cache 是否“命中”的判断，这样就减少了 cache 更新的次数。cache 可以用 MODE2 寄存器设置为禁止或冻结(不更新)。

ADSP21160 具有丰富的条件执行指令。

3．地址产生器和总线

两套数据地址产生器 DAG1、DAG2 分别指向 DM 和 PM 区。每套 DAG 都配有 8 个地址寄存器(DAG1 为 I0～I7，DAG2 为 I8～I15)和 8 个地址修改寄存器(DAG1 为 M0～M7，DAG2 为 M8～M15)。利用地址寄存器，ADSP21160 可以完成指令执行前的地址预修改和执行完毕后的地址后修改操作。

ADSP21160 还为每套 DAG 提供了 8 对循环基址寄存器(DAG1 为 B0～B7，DAG2 为 B8～B15)和循环长度寄存器(DAG1 为 L0～L7，DAG2 为 L8～L15)，用以在同一时间段内进行 8 种循环寻址。ADSP21160 还支持两套 DAG 地址的位反序寻址(利用 I0 或 I8)。ADSP21160 的程序只能放在 PM 区中，而数据可以放在 DM 或 PM 区中。

DAG 的操作模式受 MODE1 寄存器中位的控制。

DAG2 产生的 PM 地址总线宽度为 32 位，可以访问最多达 4 GB 的程序/数据混合存储区。

DAG1 产生的 DM 地址总线宽度也为 32 位，可以访问最多达 4 GB 的数据存储区。

PM 数据总线宽 64 位，用以传送 48 位字长的指令和 64 位长字。

DM 数据总线宽 64 位，用以传送 32 位正常字、40 位扩展精度字和 64 位长字。DM 数据总线可以与 DSP 的任何寄存器作数据传递。还有一个特殊的 PX 总线交换寄存器可以在 64 位 PM 数据总线和 64 位 DM 数据总线或 40 位寄存器组之间传递数据。

4．寄存器总结

ADSP21160 提供了大量的寄存器，分别完成特定的功能。将这些寄存器按其功能总结如表 2.96 所示。

表 2.96　ADSP21160 的寄存器总结

寄存器类型		寄存器名	功　能
通用存器(ureg)	数据寄存器组(dreg)	R0～R15(F0～F15)	运算核 PEx 中的寄存器组定点(浮点)
		S0～S15(SF0～SF15)	运算核 PEy 中的寄存器组定点(浮点)
	程序控制寄存器	PC	程序计数器
		PCSTK	24 位的 PC 栈顶地址(最高地址)
		PCSTKP	PC 堆栈指针
		FADDR	取指地址(只读)
		DADDR	译码地址(只读)
		LADDR	循环(Loop)终止地址；循环地址堆栈的栈顶
		CURLCNTR	当前循环计数器；循环(Loop)计数器堆栈的栈顶
		LCNTR	下一层 Loop 的循环计数值

续表(一)

寄存器类型		寄 存 器 名	功　能
通用存器(ureg)	数据地址产生寄存器	I0～I7	DAG1 的地址寄存器
		M0～M7	DAG1 的地址修改寄存器
		L0～L7	DAG1 的循环寻址长度寄存器
		B0～B7	DAG1 的循环寻址基址寄存器
		I8～I15	DAG2 的地址寄存器
		M8～M15	DAG2 的地址修改寄存器
		L8～L15	DAG2 的循环寻址长度寄存器
		B8～B15	DAG2 的循环寻址基址寄存器
	总线交换寄存器	PX1	PM↔DM 总线交换寄存器 1(32 位)
		PX2	PM↔DM 总线交换寄存器 2(32 位)
		PX	64 位的 PX1 和 PX2 的组合，PX2 为高 32 位，PX1 为低 32 位
	定时器	TPERIOD	定时器的周期
		TCOUNT	定时器的计数值
	系统寄存器(sreg)	MODE1	模式控制寄存器 1
		MODE2	模式控制寄存器 2
		IRPTL	中断锁存寄存器
		IMASK	中断屏蔽寄存器
		IMASKP	中断屏蔽指针，指向嵌套的中断
		MMASK	模式屏蔽寄存器
		FLAGS	FLAG 管脚输入/输出状态值
		LIRPTL	链路口中断锁存、屏蔽的指针
		ASTATx	运算核 PEx 的运算状态标志、位测试标志
		ASTATy	运算核 PEy 的运算状态标志、位测试标志
		STKYx	运算核 PEx 的附加运算状态标志、堆栈状态标志
		STLYy	运算核 PEy 的附加运算状态标志、堆栈状态标志
		USTAT1～4	用户状态寄存器 1～4
乘法结果寄存器		MR，MR0，MR1，MR2	乘法结果寄存器，包括 MRF 和 MRB
		MRF，MR0F，MR1F，MR2F	乘法结果寄存器(前台)
		MRB，MR0B，MR1B，MR2B	乘法结果寄存器(后台)
IOP 寄存器(存储器映射寄存器)	系统控制	SYSCON	系统设置寄存器
		SYSTAT	系统状态寄存器
		WAIT	等待寄存器
		VIRPT	多处理器矢量中断寄存器
		MSGR0～MSGR7	消息寄存器
		BMAX	总线最大超时寄存器
		BCNT	总线超时计数器
		ELAST	上一次访问的外部地址
	DMA	EPB0～EPB3	外部口 FIFO 缓存
		DMAC10～DMAC13	DMA 控制寄存器(对应 EPB0～EPB3)
		DMASTAT	DMA 标志寄存器
		IIx,IMx,Cx,CPx,GPx,DBx,DAx	DMA 通道 x (x=0～9)参数寄存器
		IIx,IMx,Cx,CPx,GPx,EIx,EMx,Ecx	DMA 通道 x(x=10～13)参数寄存器

续表(二)

寄存器类型		寄 存 器 名	功　能
IOP 寄存器(存储器映射寄存器)	链路口	LBUF0～LBUF5	链路口缓冲
		LCTL0, LCTL1	链路口 LBUF 控制寄存器
		LCOM	链路口通用控制寄存器
		LAR	链路口指定寄存器
		LSRQ	链路口服务请求寄存器
		LPATH1～LPATH3	多处理器链路路径(网络)
		LPCNT	链路路径计数器(网络)
		CNST1,CNST2	链路常数(网络)
	串口	STCTLx,SRCTLx,TXx,RXx, TDIVx,RDIVx,MTCSx,MRCSx,MTCCSx, MRCCSx,SPATHx,KEYWDx, KEYMASKx	串口 x 寄存器 x=0，1

对最常用的寄存器，ADSP21160 提供了两套相同的寄存器组，以便在调用程序和被调用程序之间作上下文切换使用。这样的主寄存器/备用寄存器包括地址产生寄存器(DAG1、DAG2)、MR 寄存器和数据寄存器 R0～R15(PEx)、S0～S15(PEy)。这些备用寄存器的选择由 MODE1 寄存器的位控制。

控制与标志寄存器设置处理器核如何工作，并指示处理器核的状态。这些寄存器被归为系统寄存器(Sreg)类。控制与标志寄存器包括：模式控制寄存器 1(MODE1)、模式屏蔽寄存器(MMASK)、模式控制寄存器 2(MODE2)、运算状态标志寄存器(ASTATx 和 ASTATy)、辅助运算状态标志寄存器(STKYx 和 STKYy) 和用户定义的状态寄存器(USTAT1～4)。这些寄存器分别如表 2.97～2.100 所示。

表 2. 97　MODE1 的位定义

位	名　称	定　义
0	BR8	I8(DAG2)的位反序寻址使能：1＝使能，0＝关闭
1	BR0	I0(DAG1)的位反序寻址使能：1＝使能，0=关闭
2	SRCU	MR 备用寄存器选择：1＝选择备用，0＝选择主寄存器
3	SRD1H	DAG1 高 4 寄存器(7～4)备用选择：1＝选择备用，0＝选择主寄存器
4	SRD1L	DAG1 低 4 寄存器(3～0)备用选择：1＝选择备用，0＝选择主寄存器
5	SRD2H	DAG2 高 4 寄存器(15～12)备用选择：1＝选择备用，0＝选择主寄存器
6	SRD2L	DAG2 低 4 寄存器(11～8)备用选择：1＝选择备用，0＝选择主寄存器
7	SRRFH	寄存器组(R15～R8)备用选择：1＝选择备用，0＝选择主寄存器
9、8	—	保留
10	SRRFL	寄存器组(R7～R0)备用选择：1＝选择备用，0＝选择主寄存器
11	NESTM	中断嵌套使能：1＝使能，0＝禁止
12	IRPTEN	全局中断使能：1＝使能，0＝禁止
13	ALUSAT	ALU 定点饱和处理：1＝溢出时，采取饱和处理，0＝直接从高 32 位取
14	SSE	短字符号扩展：1＝高 16 位符号扩展，0＝高 16 位填 0
15	TRUNC	1＝向 0 截断结果，0＝向最接近的数截断结果
16	RND32	1＝把浮点数截取到 32 位，0＝把浮点数截取到 40 位

续表

位	名　称	定　　义
18、17	CESL	指示 DSP 是否控制着外部总线：00=DSP 控制总线，其它=DSP 没有控制总线
20、19	—	保留
21	PEYEN	运算核 PEy 使能：1＝使能 PEy(SIMD 模式)，0＝禁止 PEy(SISD 模式)，当禁止时，PEY 进入低功耗状态
22	BDCST9	寄存器广播使能(通过地址寄存器 I9)：1＝使能，0＝禁止 当使能时，通过地址寄存器 I9 把 PM 区的数据广播到 PEx 和 PEy 相对应的数据寄存器中
23	BDCST1	寄存器广播使能(通过地址寄存器 I1)：1＝使能，0＝禁止 当使能时，通过地址寄存器 I1 把 DM 区的数据广播到 PEx 和 PEy 相对应的数据寄存器中
24	CBUFEN	循环寻址使能：1＝使能，0＝禁止
31～25	—	保留

注：MODE1 在复位后的值为 0000 0000h。

模式屏蔽寄存器 MMASK 与 MODE1 中的位一一对应。当有中断或指令 PUSH Sts 时，ADSP21160 把 MODE1 的内容放入状态堆栈中，并把 MMASK 的值加载到 MODE1 中。

表 2.98　MODE2 的位定义

位	名　称	功　　能
0	IRQ0E	1= $\overline{\text{IRQ0}}$ 沿触发，0= $\overline{\text{IRQ0}}$ 电平触发
1	IRQ1E	1= $\overline{\text{IRQ1}}$ 沿触发，0= $\overline{\text{IRQ1}}$ 电平触发
2	IRQ2E	1= $\overline{\text{IRQ2}}$ 沿触发，0= $\overline{\text{IRQ2}}$ 电平触发
3	—	保留
4	CADIS	指令 cache 禁止：1＝禁止，0＝使能
5	TIMEN	定时器使能：1＝使能，0＝禁止
6	BUSLK	外部总线锁定(对多处理器)：1＝锁定，0＝未锁定
14～7	—	保留
15	FLG0O	FLG0 输出、输入选择：1＝输出，0＝输入
16	FLG1O	FLG1 输出、输入选择：1＝输出，0＝输入
17	FLG2O	FLG2 输出、输入选择：1＝输出，0＝输入
18	FLG3O	FLG3 输出、输入选择：1＝输出，0＝输入
19	CAFRZ	指令 cache 冻结(不更新)：1＝冻结，0＝允许新指令输入
20	IIRAE	检测非法 IOP 寄存器的访问使能：1＝允许检测，0＝禁止检测 当置位后，如果 DSP 发现一个非法的访问，就会置位 STKYx 寄存器的 IIRA 位
21	U64MAE	检测非连续的 64 位长字存储器空间的访问使能：1＝使能检测，0＝禁止检测 当置位后，如果 DSP 发现一个非连续的长字空间访问，就会置位 STKYx 寄存器的 U64MA 位
24～22	—	保留
27～25	PID2～0	处理器 ID 号
29、28	—	芯片版本号
31、30	PID4～3	处理器 ID 号

注：MODE2 在复位后的值为 xx00 0000h。

表 2.99　算术标志寄存器(ASTATx/ASTATy)的位定义

位	名 称	功　　能
0	AZ	ALU 为 0 或浮点下溢出，1＝ALU 结果为 0 如果 ALU 浮点结果下溢出，就会置位 STKYx/STKYy 的 AUS 位，同时置位此位
1	AV	ALU 浮点上溢出(或定点溢出)，1＝溢出，对于定点，DSP 置位此位和 STKYx/STKYy 的 AOS 位；对于浮点，DSP 置位此位和 STKYx/STKYy 的 AVS 位
2	AN	ALU 结果为负(浮点或定点)，1＝结果为负，0＝结果为正
3	AC	ALU 定点进位，1＝进位，0＝未进位
4	AS	ALU 操作数 X 的符号(对于 ABS 和 MANT 指令)，1＝负，0＝正
5	AI	ALU 浮点无效数操作，1＝无效，0＝有效，当进行下述操作时，置位此位和 STKYx/STKYy 的 AIS 位：NAN 输入数；加一个正无穷大数；减一个负无穷大数；当饱和模式没有设置，而浮点向定点转化时溢出；当饱和模式没有设置时，对无穷大数操作
6	MN	乘法器结果为负(定点或浮点)，1＝负，0＝非负
7	MV	乘法器结果上溢出(浮点或定点)，1＝上溢出。对浮点结果溢出，置位此位和 STKYx/STKYy 的 MVS 位；对定点结果溢出，置位此位和 STKYx/STKYy 的 MOS 位
8	MU	乘法器结果下溢出(浮点或定点)，1＝下溢出。对浮点结果溢出，置位此位和 STKYx/STKYy 的 MUS 位；对定点结果溢出，置位此位和 STKYx/STKYy 的 MOS 位
9	MI	乘法器浮点无效数操作，1＝无效，当有下述操作时，置位此位和 STKYx/STKYy 的 MIS 位：NAN 输入数；无穷大和一个零输入数
10	AF	ALU 浮点操作，1＝浮点操作，0＝定点操作
11	SV	移位器结果溢出，1＝溢出
12	SZ	移位器结果为 0，1＝结果为 0，如果 DSP 执行一个超过 32 位定点域的位测试，也会置位 SZ
13	SS	移位器输入符号，1＝输入数为负，0＝输入数为正
17～14	—	保留
18	BTF	系统寄存器的位测试标志
23～19	—	保留
31～24	CACC	8 次比较累计位，bit31 存放最后一次比较结果，bit24 存放最早的比较结果，1=X 操作数大于 Y 操作数，0=Y 操作数大于 X 操作数

注：ASTATx/ASTATy 在复位后的值为 0000 0000h，每个运算核有它自己的 ASTAT 寄存器。ASTATx 标志 PEx 的操作，ASTATy 标志 PEy 的操作。

表 2.100　辅助标志寄存器(STKYx/STKYy)的位定义

位	名 称	功　　能
0	AUS	ALU 浮点下溢出
1	AVS	ALU 浮点上溢出
2	AOS	ALU 定点溢出
4、3	—	保留
5	AIS	ALU 浮点无效数操作
6	MOS	乘法器定点溢出
7	MVS	乘法器浮点上溢出
8	MUS	乘法器浮点下溢出
9	MIS	乘法器浮点无效数操作
16～10	—	保留
17	CB7S	DAG1 循环缓冲 7 溢出

续表

位	名 称	功 能
18	CB15S	DAG1 循环缓冲 15 溢出
19	IIRA	非法的 IOP 寄存器访问
20	U64MA	非连续的 64 位长字存储器空间访问
21	PCFL	PC 堆栈满
22	PCEM	PC 堆栈空
23	SSOV	状态堆栈满(MODE1 和 ASTATx/ASTATy)
24	SSEM	状态堆栈空
25	LSOV	循环堆栈满(循环地址和计数器)
26	LSEM	循环堆栈空
31～27	—	保留

注：STKYx/STKYy 在复位后的值为 0000 0000h，每个运算核有它自己的 STKY 寄存器。STKYx 标志 PEx 的操作和程序控制器的部分堆栈状态，STKYy 标志 PEy 的操作和程序控制器的部分堆栈状态。

用户定义的寄存器 USTAT1～4，在 PEx 中为 USTAT1、USTAT3，而 PEy 中相对应的为 USTAT2、UASTA4。这些寄存器复位后的值为 0000 0000h。程序中可用位指令(SET、CLEAR、TEST 等)来操作这些寄存器，也可以用作低开销的、一般目的标志，还可以暂存 32 位数据。

ADSP21160 增加了一个新的寄存器 FLAGS，用来存放 FLAG 管脚的输入或输出状态值，如表 2.101 所示。

表 2.101　FLAGS 寄存器的位定义

位	名 称	定 义
0	FLG0	FLAG0 值，如果置位表示 FLAG0 管脚为高，否则表示 FLAG0 管脚为低
1	FLG1	FLAG1 值，如果置位表示 FLAG1 管脚为高，否则表示 FLAG1 管脚为低
2	FLG2	FLAG2 值，如果置位表示 FLAG2 管脚为高，否则表示 FLAG2 管脚为低
3	FLG3	FLAG3 值，如果置位表示 FLAG3 管脚为高，否则表示 FLAG3 管脚为低
31～4	—	保留

IOP 寄存器是存储器映射寄存器，这些寄存器占据地址空间 00～FFh，IOP 寄存器控制着 I/O 口的操作。

因为 IOP 寄存器是存储器映射寄存器，可作为存储器来访问这些寄存器。尽管这些寄存器有存储器映射地址，但它们与 DSP 的内部存储器是分开的，总线访问也不相同。在每一时刻，只能有一套总线可以访问同一 IOP 寄存器组(如表 2.96 所示)中的一个 IOP 寄存器。

当几套总线同时对同一 IOP 寄存器组中的寄存器访问时，DSP 会按下列顺序来仲裁寄存器的访问：外部口(EP)总线访问(最高优先级)、DM 总线访问、PM 总线访问、IO 总线访问(最低优先级)。

当高优先级总线正在访问 IOP 寄存器时，低优先级总线访问同一组中的寄存器被阻止，直到高优先级总线完成对 IOP 寄存器的访问。有一个例外就是，IO 总线和 EP 总线能同时访问 DB(DMA buffer)组中的寄存器，这就使得 DMA 可以全速存取内部存储器数据。

由于 IOP 寄存器是存储器映射的，ADSP21160 的结构不允许程序直接在这些寄存器和其它存储器之间传送数据(除了部分 DMA 操作)。要读写 IOP 寄存器，程序必须用到处理器核寄存器。

下例中把 I0 指向的存储器内容写到 IOP 寄存器的 WAIT 中，注意必须要经过两步：

USTAT2＝DM(I0)

DM(WAIT)＝USTAT2

表 2.102 列出了系统配置寄存器 SYSCON 的位定义。

表 2. 102　系统配置寄存器(SYSCON)的位定义

位	名　称	功　能
0	SRST	软件复位。当程序置位此位后，ADSP21160 响应不可屏蔽 RSTI 中断(复位)，并清除此位
1	BSO	引导方式选择忽略：1=使能引导存储器，0=禁止。当置位时，ADSP21160 用 $\overline{BMS}$ 选择线(而不是 $\overline{MS3\sim0}$)通过 DMA 的通道 10 访问外部存储器。当从 8 位的引导存储器读取数据时，DSP 执行 8→48 位的打包，但向引导存储器写入数据时，不执行打包。要正确地向引导区存储器写入数据，程序必须用移位器把要传送的字节放到 64 位数据线的 32～39 位
2	IIVT	内部中断矢量表：1=强制把中断矢量表放在地址为 0004 0000h 的空间上，而不管引导模式如何，0=把中断矢量表放到由引导模式指定的位置上
3	—	保留
6～4	HPM	主机打包模式：000＝不打包，001＝16→32/64(复位值)，010＝16→48，011＝32→48，100＝32→32/64
7	HMSWF	主机打包次序：1＝高位在前，0＝低位在前
8	HPFLSH	主机打包状态刷新：1=此位刷新 FIFO，执行下述操作：清除 SYSTAT 寄存器中的 HPS 状态、清除 DMA 的请求计数器、清除未打包完的字
9	IMDW0	片内存储块 0 正常字数据宽度：1＝40 位，0＝32 位
10	IMDW1	片内存储块 1 正常字数据宽度：1＝40 位，0＝32 位
11	ADREDY	有效驱动 REDY 方式，1＝有效驱动，0＝漏极开路
15～12	MSIZE	外部存储器组(Bank)的大小，这些位指定 4 个外部存储器组(3～0)的大小。没有分配给这 4 个组的外部存储器是无分组区。组的大小与 MSIZE 的关系为：MSIZE＝[$\log_2$(32 位字下的组大小)]－13
16	BHD	缓冲悬挂禁止：1＝防止处理器核挂起，0＝允许处理器核挂起
18～17	EBPR	外部总线访问优先权，00＝优先级在处理器核总线和 IO 总线之间交替，01＝处理器核总线优先，10＝IO 总线优先
19	DCPR	外部口 DMA 通道(10～13)循环优先级使能：1＝使能，0＝禁止(固定优先级)
20	LDCPR	链路口 DMA 通道(4～9)循环优先级使能：1＝使能，0＝禁止(固定优先级)
21	PRROT	链路口 DMA 通道(4～9)与外部口 DMA 通道(10～13)之间的循环优先级使能：1＝使能，0＝禁止(固定优先级)
22	COD	时钟输出禁止：1＝CLKOUT 管脚输出 DSP 时钟，0＝CLKOUT 管脚为三态状态
31～23	—	保留

注：SYSCON 的地址为 000h，复位值为 10010h。

程序控制器用矢量中断地址寄存器 VIRPT 来支持多处理器矢量中断。矢量中断(VIRPTI)允许在多处理器系统中传递内部处理器命令。当一个外部处理器(主机或另一个 DSP)写一个地址到 VIRPT 寄存器中时，这个中断就会发生，并在 VIRPTI 中插入一个新的矢量地址。矢量中断地址寄存器 VIRPT 的位定义如表 2.103 所示。

表 2.103　矢量中断地址寄存器(VIRPT)的位定义

位	名　称	功　　能
23～0	VIRPTA	矢量中断地址。当一个外部处理器加载一个地址到此寄存器时，DSP 把当前状态压入堆栈，并开始执行矢量地址指向的程序
31～24	VIRPTD	矢量中断数据(可选)。这些位包含由外部处理器传给中断服务程序的数据

外部存储器等待状态与访问模式寄存器 WAIT 和系统状态寄存器 SYSTAT 的位定义分别如表 2.104 和表 2.105 所示。其它 IOP 寄存器在后面关于链路口、串口、外部口和 DMA 的介绍中再作说明。

表 2.104　外部存储器等待状态与访问模式寄存器(WAIT)的位定义

位	名　称	功　　能
1～0	EB0AM	外部存储器组 0(Bank0)的访问模式： 00=异步模式，根据 EB0WS 位中设置的等待状态值来访问外部存储器，并且需要应答信号 ACK 01=同步访问，读存储器根据 EB0WS 位中设置的等待状态值(最小为 001)，写存储器需要 0 等待状态 10=同步模式，读存储器根据 EB0WS 位中设置的等待状态值(最小为 001)，写存储器需要 1 等待状态 11=保留
4～2	EB0WS	外部存储器组 0 的等待状态： EB0W　等待状态　是否持续周期 000　0　否 001　1　否 010　2　是 011　3　是 100　4　是 101　5　是 110　6　是 111　7　是 注意：持续周期仅用于异步模式
6～5	EB1AM	外部存储器组 1 的访问模式，同上 EB0AM
9～7	EB1WS	外部存储器组 1 的等待状态，同上 EB0WS
11～10	EB2AM	外部存储器组 2 的访问模式，同上 EB0AM
14～12	EB2WS	外部存储器组 2 的等待状态，同上 EB0WS
16～15	EB3AM	外部存储器组 3 的访问模式，同上 EB0AM
19～17	EB3WS	外部存储器组 3 的等待状态，同上 EB0WS
21～20	UBAM	外部无分组区访问模式，同上 EB0AM
24～22	UBWS	外部无分组区等待状态，同上 EB0WS
27～25	PAGSZ	外部 DRAM 页大小(仅用于块 0)： 000＝256，001＝512，010＝1024，011＝2048，100＝4096，101＝8192，110＝16384，111＝32768 字
29～28	—	保留
30	HIDMA	DMA 的握手和 IDLE 使能，此位使能(置位)时，在每个 DMA 通过握手访问外部存储器后加入一个 IDLE 周期
31	—	保留

注：此寄存器在复位后的默认设置为：异步读取外部存储器模式，等待状态 7，具有一个持续周期，外部 DRAM 页为 256 字，禁止 DMA 握手和插入 IDLE。

表 2.105　系统状态寄存器(SYSTAT)的位定义

位	名　称	功　　能
0	HSTM	主机控制总线：1＝主机控制总线，0＝主机未控制总线
1	BSYN	此位指示总线逻辑是否为同步：1＝同步，0＝异步
3、2	—	保留
6～4	CRBM	在多处理器系统中，这些位指示当前总线控制者的 ID 码
7	—	保留
10～8	IDC	这些位指示 ADSP21160 的 ID 管脚状态
11	—	保留
12	DWPD	直接写内部存储器挂起：1=挂起，0=完成
13	VIPD	矢量中断挂起：1=挂起，0=完成
15～14	HPS	主机打包状态：00＝打包完成，01＝打包或展开的第一步(未完成打包)，10＝打包或展开的第二步(对应多步打包，未完成打包)，11＝保留
19～16	CRAT	核时钟与输入时钟比，这些位指示 ADSP21160 的 CLK_CFG3～0 管脚的状态
31～20	—	保留

2.4.3　存储器组织

1．存储器映射

ADSP21160 采用多总线结构，其存储空间分为三部分：内部存储器空间、多处理器存储器空间和外部存储器空间。图 2.16 为 ADSP21160 的存储空间映射图。

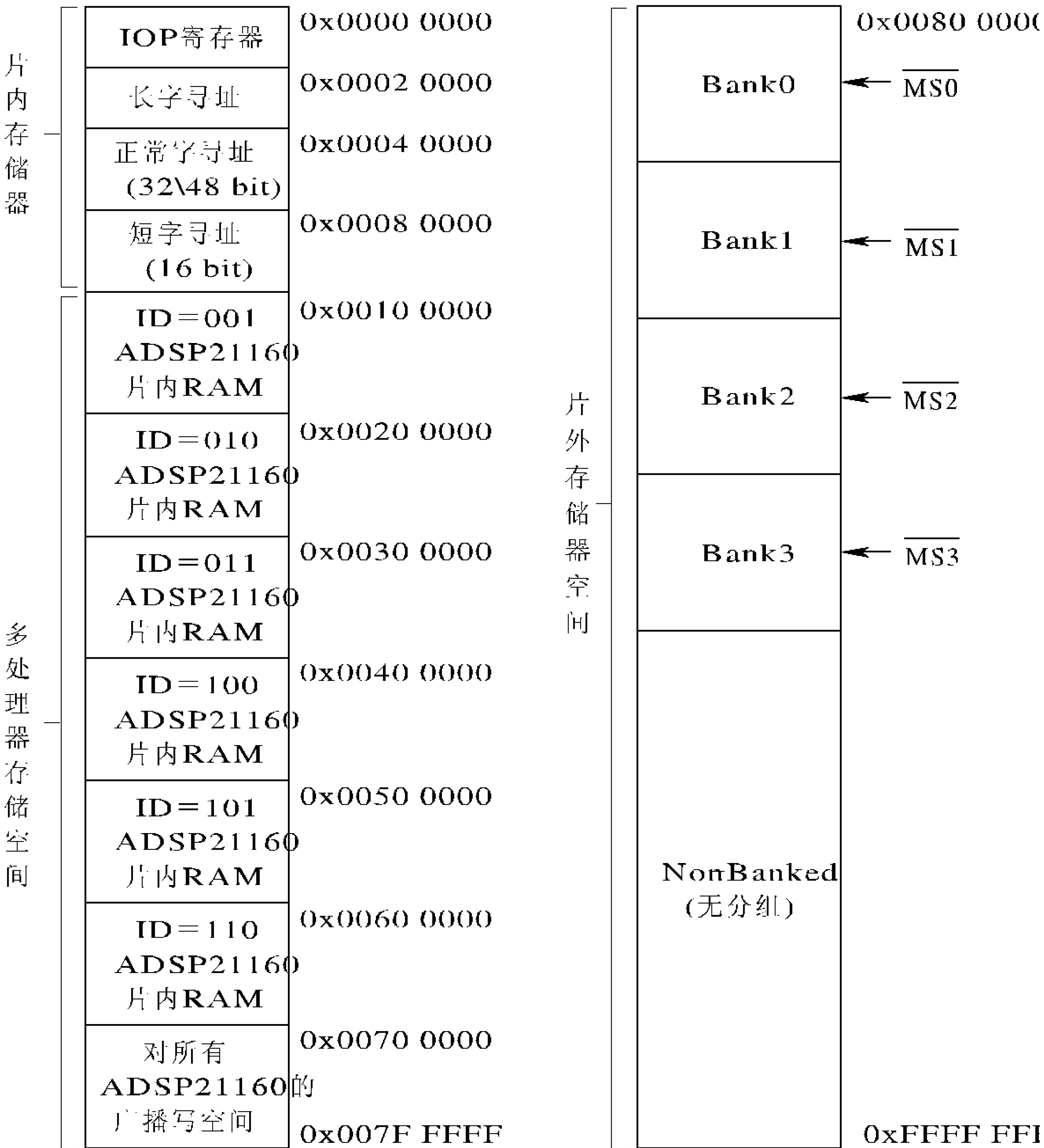


图 2.16　ADSP21160 的存储空间映射图

2．片内存储器

ADSP21160 具有 8 Mb 的片内存储器寻址空间，但当前只提供了 4 Mb SRAM，它等量的分成两个存储块：0 和 1(分别由 DAG1，DAG2 控制)，可以作为 64 位、48 位、32 位或 16 位数据进行访问。存储块 0 和存储块 1 在不同字宽下对应的地址范围是不同的：

当设置为 64 bit 长字时，两存储块分别占据 0002 0000h～0002 7FFFh 和 0002 8000h～0002 FFFFh 存储空间；

当设置为 32 bit 正常字时，两存储块分别占据 0004 0000h～0004 FFFFh 和 0005 0000h～0005 FFFFh 存储空间；

当设置为 48 bit 字时，两存储块也分别占据 0004 0000h～0004 FFFFh 和 0005 0000h～0005 FFFFh 存储空间，但两存储块末段的部分存储空间不可用；

当设置为 16 bit 短字时，两存储块分别占据 0008 0000h～0009 FFFFh 和 000A 0000h～000B FFFFh 存储空间。

事实上，无论设置为 64 bit、48 bit、32 bit 还是 16 bit 字宽，尽管地址空间不同，但它们都对应同一物理存储空间。

ADSP21160 有三条内部总线与片内存储器相连，它们是 PM 总线、DM 总线、I/O 总线，并且它们的数据总线都是 64 位，地址总线都是 32 位，在同一周期内，三条总线都可以对片内存储器进行访问。处理器核对片内存储器的访问与 I/O 处理器对片内存储器的访问是独立和透明的，即在同一周期内，处理器核和 I/O 处理器都可以访问片内存储器。

I/O 处理器(简称 IOP)的寄存器是存储器映射寄存器，它们占据 0000 0000h～0000 00FFh 地址空间范围。IOP 寄存器和长字地址空间之间的 0000 0100h～0001 FFFFh 为保留地址，不能访问，IOP 寄存器映射被组织成连续的 32 位地址。对这部分地址空间的长字访问(指令带后缀“LW”)，返回到数据总线低 32 位上的是所指地址的内容，数据总线的高 32 位为 0，而不是下一个寄存器的内容。一个例外是在 SIMD 模式下，访问外部口数据缓冲区、PX 与外部存储器或链路口数据缓冲区的数据交换，返回的是两个连续的 32 位值。

3．外部存储器

ADSP21160 通过外部总线接口的 32 位地址线和 64 位数据线以及有关控制信号线访问片外的程序代码或数据，4 G 寻址空间的片外存储器分成五组(Bank)：组 0、组 1、组 2、组 3 和无分组空间，前四组可以用 ADSP21160 的输出信号线 $\overline{\text{MS0～3}}$ 分别选中，每组存储空间的大小通过 SYSCON 寄存器的 MSIZE 位段来设置。

外部存储器可以是 16、32、48、64 位，ADSP21160 的 DMA 控制器可以自动打包外部数据以适合内部的字宽。核对外部的访问可以是 32、48、64 位。

4．多处理器存储空间

ADSP21160 支持多达 6 片的总线共享连接，并且无需任何外加控制逻辑。每片 DSP 根据其标识码管脚(ID2～0)的电平设置二进制编号：001b～110b(单片独立工作时为 000b)。当每片 DSP 的 PM/DM 总线指向其自身所处的存储空间时，指令/数据的访问表现为片内访问。当指向其它 DSP 所在存储空间时，表现为多处理器存储共享形式。当指向片外存储器时，则为片外存储器访问。所有共享总线的 ADSP21160，其外部总线的地址/数据线连在一起，在每个时刻只有一片 ADSP21160(称主处理器)对此总线有控制权，并可以对其它

ADSP21160(称从处理器)的片内存储空间甚至控制寄存器进行访问，还可以向其它所有 ADSP21160 的片内存储器同时写同样内容(广播写)。这一切都靠 ADSP21160 的片内总线仲裁逻辑和一些多处理器控制握手信号线来完成。当各 ADSP21160 都访问各自片内存储器或仅有一个 ADSP21160 访问片外存储器时，各处理器实际上是独立工作的。

2.4.4 中断

ADSP21160 有多种类型中断：外部中断 5 个，即仿真器中断、复位中断和 3 个外部管脚中断信号；内部中断有 30 个，即定时器中断(2 个)，DMA 控制器中断(14 个)，循环寻址缓冲区溢出中断(2 个)，非法输入中断(1 个)，堆栈溢出(1 个)，运算出错(4 个)，多处理器矢量中断(1 个)，链路服务请求(1 个)及用户定义的软中断(4 个)。表 2.106 列出了这些中断对应的中断矢量地址和相应的寄存器控制位。

表 2.106 ADSP21160 中断矢量地址与 IRPTL/LIRPTL、IMASK 的关系

寄存器	IRPTL/IMASK 或 LIRPTL 的相应位	中断矢量地址(Hex)	中断名称	功　能
IRPTL/IMASK	0	00	EMUL	仿真器中断(只读，不可屏蔽)，有最高优先级
IRPTL/IMASK	1	04	RSTI	复位(只读，不可屏蔽)
IRPTL/IMASK	2	08	IICDI	发现非法输入
IRPTL/IMASK	3	0C	SOVFI	状态/循环/模式堆栈溢出或 PC 堆栈满
IRPTL/IMASK	4	10	TMZHI	定时器＝0(高优先级)
IRPTL/IMASK	5	14	VIRPTI	矢量中断
IRPTL/IMASK	6	18	IRQ2I	$\overline{IRQ2}$ 触发
IRPTL/IMASK	7	1C	IRQ1I	$\overline{IRQ1}$ 触发
IRPTL/IMASK	8	20	IRQ0I	$\overline{IRQ0}$ 触发
IRPTL/IMASK	9	24	—	保留
IRPTL/IMASK	10	28	SPR0I	DMA 通道 0～SPORT0 接收
IRPTL/IMASK	11	2C	SPR1I	DMA 通道 1～SPORT1 接收
IRPTL/IMASK	12	30	SPT0I	DMA 通道 2～SPORT0 发送
IRPTL/IMASK	13	34	SPT1I	DMA 通道 3～SPORT1 发送
LIRPTL	0(锁存)/16(屏蔽)	38	LP0I	DMA 通道 4～LBUF0
LIRPTL	1(锁存)/17(屏蔽)	3C	LP1I	DMA 通道 5～LBUF1
LIRPTL	2(锁存)/18(屏蔽)	40	LP2I	DMA 通道 6～LBUF2
LIRPTL	3(锁存)/19(屏蔽)	44	LP3I	DMA 通道 7～LBUF3
LIRPTL	4(锁存)/20(屏蔽)	48	LP4I	DMA 通道 8～LBUF4
LIRPTL	5(锁存)/21(屏蔽)	4C	LP5I	DMA 通道 9～LBUF5
IRPTL/IMASK	15	50	EP0I	DMA 通道 10～外部口 EPB0
IRPTL/IMASK	16	54	EP1I	DMA 通道 11～外部口 EPB1
IRPTL/IMASK	17	58	EP2I	DMA 通道 12～外部口 EPB2
IRPTL/IMASK	18	5C	EP3I	DMA 通道 13～外部口 EPB3
IRPTL/IMASK	19	60	LSRQI	链路服务请求
IRPTL/IMASK	20	64	CB7I	循环缓冲 7 溢出

续表

寄存器	IRPTL/IMASK 或 LIRPTL 的相应位	中断矢量地址(Hex)	中断名称	功　能
IRPTL/IMASK	21	68	CB15I	循环缓冲 15 溢出
IRPTL/IMASK	22	6C	TMZLI	定时器＝0(低优先)
IRPTL/IMASK	23	70	FIXI	定点溢出
IRPTL/IMASK	24	74	FLTOI	浮点上溢出
IRPTL/IMASK	25	78	FLTUI	浮点下溢出
IRPTL/IMASK	26	7C	FLTII	浮点无效
IRPTL/IMASK	27	80	SFT0I	用户软中断 0
IRPTL/IMASK	28	84	SFT1I	用户软中断 1
IRPTL/IMASK	29	88	SFT2I	用户软中断 2
IRPTL/IMASK	30	8C	SFT3I	用户软中断 3
IRPTL/IMASK	31	90	—	保留(最低优先级)
IRPTL/IMASK	14	—	—	链路中断汇总

注：中断矢量表从片内的 0004 0000h 或片外的 0080 0000h 处开始。

表 2.106 的中断优先权自上而下递降。为了灵活使用定时器，为其定义了两个优先权不同的中断矢量。中断矢量地址是偏移量，当使用片内存储器时，中断矢量基址为 0004 0000h；当使用片外存储器时，基址为 0080 0000h。

除了复位中断和 EMUL 中断是不可屏蔽的外，其余所有中断都是可屏蔽的。由 IMASK 和 LIRPTL(链路中断)寄存器的相应位来设置屏蔽或使能。在 MODE1 寄存器中还有一个全局中断屏蔽/使能位(IRPTEN)，控制所有可屏蔽中断。这些寄存器的相应位置位表示此中断使能，否则表示屏蔽。

如果 DSP 识别一个中断的到来，它就把此中断锁存到(置位)中断锁存标志寄存器 IRPTL 或 LIRPTL(链路口中断)的相应位(如表 2.106 所示)。在中断服务程序的执行过程中，DSP 在每一周期都清除此位，以防止在中断服务程序的执行过程中这个中断又重新锁存一次。当服务程序返回(RTI)时，程序控制器才停止清除对应的锁存位。这样，此中断如果再次到来，DSP 就会再去响应它。如果需要在中断服务程序的执行过程中允许此中断再次发生，就必须通过在中断服务程序中加入 JUMP(CI)指令清除中断标志，同时把此中断服务程序变为一个子函数。在函数返回时必须用指令 RTS(LR)。

中断嵌套允许一个中断服务程序正在执行的过程中响应另一个高优先级中断，MODE1 寄存器的 NESTM 位决定使能还是禁止中断嵌套。

对所有外部中断和定时器中断，ADSP21160 自动将 MODE1 压入堆栈保存，其压栈操作和中断服务调用并行完成，并为这样的快速中断服务提供了四级可嵌套调用。而其它寄存器，或其它中断服务时要保存的寄存器则要用指令来完成压栈操作。

2.4.5　片内外设资源

1．直接存储器访问(DMA)

直接存储器访问(DMA)可以承担数据传输任务而无需处理器核的干预，从而提高了程序执行的效率。ADSP21160 提供了 14 个 DMA 通道，数据传输由 DMA 控制器控制，并自

动完成不同字宽数据格式间的打包和展开，还能自动初始化 DMA 通道以进行下一个数据块传送(链式 DMA)。这 14 个 DMA 通道及其控制寄存器、参数寄存器和缓冲寄存器如表 2.107 所示。

表 2.107　DMA 通道及其控制寄存器、参数寄存器和缓冲寄存器

DMA 通道	控制寄存器	参数寄存器	缓冲寄存器	描　述
0	SRCTL0	II0，IM0，C0，CP0，GP0，DB0，DA0	RX0	串口 0 接收
1	SRCTL1	II1，IM1，C1，CP1，GP1，DB1，DA1	RX1	串口 1 接收
2	STCTL0	II2，IM2，C2，CP2，GP2，DB2，DA2	TX0	串口 0 发送
3	STCTL1	II3，IM3，C3，CP3，GP3，DB3，DA3，	TX1	串口 1 发送
4	LCTL0 LAR LCOM	II4，IM4，C4，CP4，GP4，DB4，DA4	LBUF0	链路 LBUF0
5		II5，IM5，C5，CP5，GP5，DB5，DA5	LBUF1	链路 LBUF1
6		II6，IM6，C6，CP6，GP6，DB6，DA6	LBUF2	链路 LBUF2
7	LCTL1 LAR LCOM	II7，IM7，C7，CP7，GP7，DB7，DA7	LBUF3	链路 LBUF3
8		II8，IM8，C8，CP8，GP8，DB8，DA8	LBUF4	链路 LBUF4
9		II9，IM9，C9，CP9，GP9，DB9，DA9	LBUF5	链路 LBUF5
10	DMAC10	II10，IM10，C10，CP10，GP10，EI10，EM10，EC10	EPB0	外部口 FIFO 缓冲 0
11	DMAC11	II11，IM11，C11，CP11，GP11，EI11，EM11，EC11	EPB1	外部口 FIFO 缓冲 1
12	DMAC12	II12，IM12，C12，CP12，GP12，EI12，EM12，EC12	EPB2	外部口 FIFO 缓冲 2
13	DMAC13	II13，IM13，C13，CP13，GP13，EI13，EM13，EC13	EPB3	外部口 FIFO 缓冲 3

ADSP21160 还有两对外部 DMA 请求/应答信号线，即 $\overline{\text{DMAR1}}$/$\overline{\text{DMAG1}}$ 和 $\overline{\text{DMAR2}}$/$\overline{\text{DMAG2}}$，分别对 DMA 通道 11 和通道 12 进行外部握手控制。

1) DMA 寄存器

DMA 寄存器包括控制寄存器、参数寄存器、数据缓冲寄存器和状态寄存器 DMASTAT。不同 DMA 通道的控制寄存器、参数寄存器、缓冲寄存器有所不同(如表 2.107 所示)。

DMA 通道的 DMA 参数寄存器如表 2.108 所示。

表 2.108　DMA 参数寄存器(x 代表 DMA 通道号 0～9)

参数寄存器名	功　能
IIx	数据源/目的地址
IMx	每一数据元素传送完成后的地址调整量(按字宽)
Cx	传送数目
CPx	链式 DMA 链指针(指向下次 DMA 参数存放地址)
GPx	通用寄存器，也可用于二维 DMA
EIx	外部总线地址(仅对外部总线 DMA)
EMx	外部总线地址调整量(仅对外部总线 DMA)
ECx	外部总线 DMA 传送数目(仅对外部总线 DMA)
DBx	通用寄存器，也可用于二维 DMA(仅对串口/链路口)
DAx	通用寄存器，也可用于二维 DMA(仅对串口/链路口)

DMA 控制寄存器用来配置 DMA 通道操作，包括 DMA 使能、链式 DMA(自动初始化)使能、传输方向、数据字宽以及其它一些配置。下面针对不同外设的 DMA 通道再分别介绍。

DMA 状态寄存器 DMASTAT 的位 0～13 对应于 DMA 通道 0～13 的激活状态：1=激活；0=未激活。位 16～29 分别对应 DMA 通道 0～13 的链式 DMA 状态，1=链式 DMA 使能，0=链式 DMA 被禁止。

下面只介绍 4 个外部口(EPB0～3)DMA 通道的控制寄存器(如表 2.109 所示)。关于串口和链路口 DMA 通道的设置方法，在本节后面对串口和链路口的介绍中再作说明。

表 2.109　DMACx(x=10～13)控制寄存器的位定义

位	名　称	定　义
0	DEN	DMA 使能：1＝使能，0＝禁止
1	CHEN	链式 DMA 使能：1=使能，0=禁止
2	TRAN	DMA 方向：1＝发送，0＝接收
4～3	PS	打包状态(只读)：00＝打包完成，01＝打包或展开的第一步，10＝打包或展开的第二步，11＝保留
5	DTYPE	数据类型选择：1＝40/48 位，0＝32/64 位
8～6	PMODE	打包/展开模式：000＝不打包，001＝外部 16 位↔内部 32/64 位，010＝外部 16 位↔内部 48 位，011＝外部 32 位↔内部 48 位，100＝外部 32 位↔内部 32/64 位，101＝110＝111＝保留
9	MSWF	1＝最高位先打包，0＝最低位先打包
10	MASTER	主机模式
11	HSHAKE	握手模式
12	INTIO	单字中断使能(非 DMA)
13	EXTERN	外部握手模式
14	FLSH	刷新外部口 FIFO 缓冲(置位)，当 FLSH＝1 时进行如下操作：清除 FS 和 PS 标志位、清除 FIFO 缓冲和 DMA 请求计数器、清除任何未完成的打包字
15	PRIO	选择外部口总线访问优先级：1＝高优先，0＝低优先
17～16	FS	外部口 FIFO 缓冲状态：00＝缓冲空，01＝缓冲非满，10＝缓冲非空，11＝缓冲满
18	INT32	内部存储器 32 位传送选择：1＝32 位传送，0＝64 位传送(如果可能)
20～19	MAXBL	最大突发(Burst)传送长度：00＝突发传送禁止，01＝突发传送长度为 4，10＝11＝保留
31～21	—	保留

2) 设置 DMA 启动步骤

按下列步骤完成 DMA 通道的数据传送启动：

(1) 清除 DMA 控制寄存器的 DMA 使能位(禁止 DMA)；

(2) 设置 DMA 通道的控制寄存器和参数寄存器(源或目的地址 IIx、地址调整量 IMx 及数据元素的计数值 Cx 等)；

(3) 置位 DMA 控制寄存器中的 DMA 使能位，这样就启动了 DMA。

DMA 设置成链式方式时，一旦一个 DMA 完成，链式 DMA 将自动启动 DMA 链中的下一个 DMA。如果要启动一种新的 DMA，就必须先禁止 DEN 位，再设置 IIx、IMx、Cx，然后再使能 DEN 位。

3) DMA 通道优先权

14 个 DMA 通道的优先权基本上是固定的，从 DMA0 到 DMA13 依次递降。串口的 4 个 DMA 通道之间的优先顺序是固定的。链路口的 6 个 DMA 通道和外部口的 4 个 DMA 通道可以另行设置循环优先模式，即最新发生 DMA 传送的那个通道优先级降到最低级。SYSCON 寄存器的 DCPR 位和 LDCPR 位用于设置循环优先模式。

4) 链式 DMA

链式 DMA 使得多个不同的 DMA 可以自动初始化并依次得到执行。

当链式 DMA 使能位 DEN 和 CHEN 置位后，向 CP 寄存器写入一个非零值就可以启动链式 DMA，而向 CP 写入 0 则禁止链式 DMA。CP 寄存器的低 18 位(bit17～0)表示相对于地址 0004 0000h 的偏移量，bit18 则称为可编程控制中断 PCI。PCI=1 使得当前 DMA 传送完成后产生一个中断请求。

CP 寄存器指向链式 DMA 的下一次 DMA 参数寄存器所在存储区(称 TCB)的最高地址，存放顺序按地址依次递减为：

外部口：IIx，IMx，Cx，CPx，GPx，EIx，EMx，ECx；

链路口或串口：IIx，IMx，Cx，CPx，GPx，DBx，DAx，LPATH1，LPATH2，LPATH3;

因此建立和启动链式 DMA 的步骤为：

(1) 在片内存储器建立要求的 TCB 数据块；

(2) 使能相应的 DEN 位和 CHEN 位；

(3) 将第一个 TCB 的最后一个地址(最高地址)写入 CP，即启动了链式 DMA。

5) DMA 中断

当传送计数器 Cx 减为 0 时，将产生 DMA 中断。对于主机模式的外部总线，DMA 还要求 ECx 寄存器也减为 0 时才能产生中断。要提醒用户的是，向这些 Cx 或 ECx 寄存器写入 0 并不会有中断产生。14 个 DMA 通道的中断优先级从 DMA 通道 0 到通道 13 递降。串口和外部口 DMA 中断将锁存在 IRPTL 寄存器中，并由 IMASK 的相应位控制是否使能，而链路口 DMA 中断的锁存和屏蔽位在 LIRPTL 寄存器中。对链式 DMA 来说，CPx 寄存器的 PCI 位也控制着中断的产生。

MODE1 寄存器的 IRPTEN 位控制着全局中断使能位。

6) DMA 的产生和终止

DMA 在链式/非链式模式下的启动条件不同，当下述条件之一满足时，DMA 将启动：链式 DMA 禁止时，DMA 使能位 DEN 从 0 变为 1；链式 DMA 使能且 DEN＝1 时，向 CPx

寄存器写入一个非 0 值；链式 DMA 使能，CPx 为非 0，而当前 DMA 结束时。

满足下列条件之一，DMA 将终止：计数寄存器 Cx 和 ECx 减为 0；链式 DMA 被禁止，DEN 从 1 变 0；当 DEN 为 0，且链式 DMA 使能时，通道进入了链插入模式。

7) 外部总线 DMA 的特别用法

4 个外部总线 DMA 通道实际上与 4 个外部口 FIFO 缓冲相连，即 DMA 通道 10、11、12、13 分别对应 EPB0、1、2、3。每个 EPBx(x=0～3)缓冲有 6 级 FIFO，通向两个端口，一个读端口和一个写端口。但当 DMA 使用 EPBx 时，处理器核不能访问此 EPBx。通过向 DMACx 寄存器的 FLSH 位写 1 可以刷新此 EPBx。

与外部总线 DMA 通道 11、12 相关的有两套 DMA 控制信号线：$\overline{\text{DMAR1/2}}$ (DMA 请求)和 $\overline{\text{DMAG1/2}}$ (DMA 确认)。DMA 控制寄存器的 MASTER 位、HSHAKE 位、EXTERN 位设置这两个通道的 DMA 模式。

8) 二维 DMA

串口和链路口 DMA 通道 0～9 支持二维 DMA。通过设置相应串口控制寄存器的 D2DMA 位或链路 LCTL0～1 控制寄存器的 LxDMA2D 位就可以设置为二维 DMA 模式，即以行主模式(先 X 方向，后 Y 方向)访问一个二维阵列元素，这时 DMA 通道参数寄存器的功能有所改变，如表 2.110 所示。

表 2.110　二维 DMA 参数寄存器

寄存器	功　能
IIx	地址，初始值为二维阵列起始地址，每一数据元素传送完后加 X 调整量，当 X 计数为 0 时，加 Y 调整量
IMx	X 地址调整量，每一数据元素传送后，地址在 X 方向的调整值
Cx	X 计数，初始值为 X 的初始计数值，每次传送减 1
CPx	指向下一个 DMA 参数所存放的内部存储器的最高地址(链式 DMA)
DBx	Y 地址调整量，当 X 计数减为 0 时，把此值加到地址 IIx 上
GPx	Y 计数，Y 方向的元素个数，相当于行数，每行传送完后(Cx → 0)减 1
DAx	X 初始计数值，二维阵列在 X 方向的元素个数，当 X 计数减为 0 时，此数值就重装入 X 计数

2．多处理器共享存储总线

参阅 2.3.5 节中 ADSP2106x 的多处理器共享存储器总线介绍。

3．主机接口

参阅 2.3.5 节中 ADSP2106x 的主机接口介绍。

4．链路口

ADSP21160 提供了 6 个链路口，可以实现与其它 ADSP21160 或外围设备的点对点通信。每个链路包括 8 位双向数据线，一个双向时钟信号，一个双向确认信号。每个链路口收发数据的时钟频率最高可达到核时钟频率(核时钟频率等于 CLKIN 乘以 CLK_CFG3～0 设置的时钟比)。

链路口的管脚如下(共 10 根线)(x=0～5)：

LxDAT7～0(链路口数据线)、LxCLK(链路口时钟线)、LxACK(链路口确认线)。

LxCLK 提供异步数据传送时钟，LxACK 是握手信号。当作为发送方时，链路口驱动数据线和 LxCLK，LxACK 为输入。当作为接收方时，链路口仅驱动 LxACK。

链路口有以下特点和功能：

- 各链路口可以独立、同时地工作。
- 链路数据可以打包成 32 bit、48 bit 数据，可以被处理器核访问，可以与片内存储器作 DMA 传送。
- 外部主机可以直接访问链路。
- 具有双缓冲的发送和接收寄存器。
- 通过可编程的时钟/确认信号在链路通信时握手，每个链路口均可收/发数据，并分别由一个 DMA 通道支持。
- 利用链路连接可以组成一维到多维的各种处理器网络。

1) 链路口模式设置

寄存器 SYSCON、LCOM、LAR 和 LCTL0～1 的位对链路口的操作模式进行控制。SYSCON 的位定义如表 2.102 所示，它们控制链路口 DMA 的优先级操作。

链路 LBUF 控制寄存器为 LCTL0～1。为了避免引起虚假中断，在修改 LCTL0～1 寄存器前，程序必须先屏蔽掉链路服务请求寄存器 LSRQ。LCTL0～1 的位定义如表 2.111 所示。LCOM 的位定义如表 2.112 所示。

表 2.111　链路 LBUF 控制寄存器(LCTL0～1)的位定义

寄存器	位	位	名　称	定　　义
LCTL0	9～0 LBUF0 控制位	0	L0EN	链路 LBUF0 使能。当禁止时，ADSP21160 清除相应的 L0STAT 和 L0RERR 位
		1	L0DEN	LBUF0 的 DMA 使能
		2	L0CHEN	LBUF0 的链式 DMA 使能
		3	L0TRAN	LBUF0 的传送方向，1＝发送，0＝接收
		4	L0EXT	LBUF0 的字扩展大小，1＝48 位，0＝32 位
		6～5	L0CLKD	LBUF0 传送时钟的除数因子，LBUF0 的传送时钟等于核时钟频率除以 L0CLKD 的值，但 L0CLKD＝0 时，应除以 4
		7	L0DMA2D	LBUF0 的二维 DMA 使能
		8	L0PDRDE	禁止对链路口 0 的 L0CLK、L0ACK、L0DAT7～0 内部下拉电阻，1＝禁止，0＝使能(此位仅对链路口 0 有效，与 LBUF0 是否指定此链路口无关)
		9	L0DPWID	链路口 0 的数据宽度(此位仅对链路口 0 有效，与 LBUF0 是否指定此链路口无关)。1＝8 位，0＝4 位。当选择 4 位数据时，使用 L0DAT3～0 来传送数据，而 L0DAT7～4 不连接(ADSP21160 把这 4 位下拉)
	19～10	LBUF1 控制，与 LBUF0 的控制位对应		
	29～20	LBUF2 控制，与 LBUF0 的控制位对应		
	31～30	保留		
LCTL1	9～0	LBUF3 控制，与 LBUF0 的控制位对应		
	19～10	LBUF4 控制，与 LBUF0 的控制位对应		
	29～20	LBUF5 控制，与 LBUF0 的控制位对应		
	31～30	保留		

表 2.112　链路口通用控制寄存器(LCOM)的位定义

位	名　称	定　义
1～0	L0STAT(1～0)	LBUF0 状态，11＝满，00＝空，10＝1 个字，01＝保留 这些位是只读的，当 L0EN 为 0 时，ADSP21160 清除这些位
3～2	L1STAT(1～0)	LBUF1 状态，同上
5～4	L2STAT(1～0)	LBUF2 状态，同上
7～6	L3STAT(1～0)	LBUF3 状态，同上
9～8	L4STAT(1～0)	LBUF4 状态，同上
11～10	L5STAT(1～0)	LBUF5 状态，同上
19～12	保留	保留
20	LMSP	使能多处理器网络
22～21	LPATHD	多处理器网络路径 LPATH 变动延迟，00＝无延迟，01＝1 个附加延迟，10＝2 个附加延迟，11＝3 个附加延迟
25～23	保留	保留
26	LRERR0	LBUF0 的接收打包出错状态，1＝未完成，0＝完成
27	LRERR1	LBUF1 的接收打包出错状态，1＝未完成，0＝完成
28	LRERR2	LBUF2 的接收打包出错状态，1＝未完成，0＝完成
29	LRERR3	LBUF3 的接收打包出错状态，1＝未完成，0＝完成
30	LRERR4	LBUF4 的接收打包出错状态，1＝未完成，0＝完成
31	LRERR5	LBUF5 的接收打包出错状态，1＝未完成，0＝完成

链路口指定寄存器 LAR 把 LBUFx 分配到链路口。LAR 的复位值为 0002 C688h，即 LBUF0 分配到 LPORT0，LBUF1 分配到 LPORT1，依此类推。链路口指定寄存器 LAR 的位定义如表 2.113 所示。

表 2.113　链路口指定寄存器(LAR)的位定义

位	名　称	定　义	备　注
2～0	A0LB	LBUF0 的指定链路口	000=链路口 0，001=链路口 1，010=链路口 2，011=链路口 3，100=链路口 4，101=链路口 5，110=保留，111=禁止
5～3	A1LB	LBUF1 的指定链路口	同上
8～6	A2LB	LBUF2 的指定链路口	同上
11～9	A3LB	LBUF3 的指定链路口	同上
14～12	A4LB	LBUF4 的指定链路口	同上
17～15	A5LB	LBUF5 的指定链路口	同上
31～18	保留	保留	

ADSP21160 有 6 个独立的链路缓冲 LBUF5～0，可以自由地定义与 6 个链路口 LPORT5～0 的连接对应关系。链路指定寄存器 LAR 用来确定 LBUF5～0 与 LPOTR5～0 间的连接关系。将一链路口指定到两个 LBUF，并禁止两个 LBUF，就可以用“自环”模式在存储器与存储器之间传送数据，这时 LxDAT7～0、LxCLK、LxACK 都不被驱动。

某链路口在两种情况下是禁止的：没有指定 LBUF，或者指定的 LBUF 被禁止。当链路口禁止时，其数据线 LxDAT7～0 及 LxCLK、LxACK 都是三态的。

2) 链路口 DMA

LBUF0～5 有 6 个专用的 DMA 通道：DMA 通道 4～9，它们之间一一对应。

LBUFx 的 DMA 使能由 LCTL1～0 寄存器的位控制。

启动链路口 DMA 的顺序如下(x=0～5, y=4～9)：

(1) 首先利用 LCTL 寄存器的 LxEN 位禁止(清 0)下面使用的 LBUFx；

(2) 由 LAR 寄存器的 AxLB 位为链路口指定此 LBUFx；

(3) 由 LCTL 寄存器的 LxEN 使能这个 LBUFx，并分别由 LxEXT 和 LxTRAN 位指定传送字宽和方向；

(4) 设置 DMA 通道参数(IIy, IMy 和 Cy)和 LCTL 控制寄存器的其它位；

(5) 置位 LCTL 寄存器的 LxDEN 后，就启动了 DMA。

LBUFx 还可以使能链式 DMA，在链式 DMA 过程中，ADSP21160 的 DMA 控制器在当前 DMA 完成后，根据 CPx 寄存器的值，由 DMA 控制器自动装入下一套放在存储器中的 DMA 参数，来建立下一个 DMA。

链路口 DMA 通道还支持二维 DMA 传送。

3) 握手信号

链路口通过握手信号 LxCLK 和 LxACK，在 ADSP21160 之间进行异步数据传送，其它遵守同样协议的外部设备也可以跟链路口进行通信。

通过设置 LCTLx 寄存器的 LxCLKD 位，使传送时钟频率为 1/2 核时钟频率，并清除 LCTLx 寄存器的 LxDPWID 位以选择 4 位字宽，就可以与 ADSP2160x 的链路口相连接，但此时 ADSP2106x 的链路口需要设置为 2 倍时钟频率。

链路口发送时钟频率可以通过 LCTLx 寄存器的 LxCLKD 位设置(1，1/2，1/3，1/4 核时钟频率)。链路口接收完全为异步的，其时钟频率可以为最高频率为核时钟的任何频率。

链路口以 8/4 位码一组的方式传送 32 位或 48 位字。发送方在时钟 LxCLK 的上升沿送出 8/4 位码，接收方利用时钟下降沿锁存 8/4 位码，并且接收方使 LxACL 有效，表示已准备好接收下一个字。在每个字开始发送时，发送方如果看到 LxACK 无效，将使 LxCLK 为高，并等待 LxACK 有效后才开始发送新字。

当链路口禁止时，LxDAT7～0、LxCLK 和 LxACK 为三态。当链路口发送使能时，数据线上是输出缓冲中的值，LxCLK 被驱动为高，LxACK 为三态。当链路口接收使能时，LxACK 被驱动为高，数据线 LxDAT7～0 和 LxCLK 为三态。

发送方驱动 LxDAT7～0、LxCLK，接收方驱动 LxACK。在链路口禁止时应对 LxPDRDE 清 0，来使 LxDAT7～0、LxCLK、LxACK 被内部下拉(50 kΩ)。应注意的是，这些信号线如果悬空，则必须用内部或外部下拉电阻。

4) 链路缓冲 LBUF

每个 LBUF 由一个内部寄存器和一个外部寄存器组成 2 级 48 位 FIFO。当 LBUF 用于发送时，内部寄存器接收从片内存储器送来的数据，外部寄存器将数据展开成 4/8 位码，并且最高位先发送。用于接收时，外部寄存器把接收数据打包并把它传递给内部寄存器，然后数据经内部寄存器自动以 DMA 方式送到片内存储器中。

当 DMA 或处理器核送来的数据占满这 2 级 FIFO 时，将送出一个“满”标志。每当一个字展开发送后，FIFO 中将空出一个位置并发出一个 DMA 请求。当 FIFO 为空时，LxCLK

就无效。

链路口字宽可以设置为 32 位或 48 位，但对于 40 位的扩展精度数据或 48 位的指令，必须设置为 48 位的字宽。

处理器核对 LBUF 直接读写比 DMA 方式的迟延要小。处理器核通过查询 LCOM 寄存器获知 LBUF 的满/空状态来访问 LBUF，这时 DMA 应禁止。当处理器核读一个空的接收 LBUF 时，它将挂起等候接收字的到来。类似地，向满的发送 LBUF 写也会挂起。为防止这类挂起可以置位 SYSCON 寄存器的 BHD 位。

主机也可以直接访问 LBUF，其字宽仅由 SYSCON 寄存器的 HPM 位(主机打包模式)决定，而与 LCTL 寄存器的 LxEXT 位无关。

5) 链路口中断

链路口中断的产生有三种类型：

• 如果 DMA 使能，当 DMA 完成(DMA 计数器为 0)时将产生中断。因为 DMA 完成时，LBUF 的接收缓冲为空，主处理器(发方)可以利用接收缓冲的两个 FIFO 空位置再发送两个字的额外信息，从处理器(收方)的中断服务程序可以读出并根据这两个字决定后续操作。

• 当某链路口 LBUF 的 DMA 禁止时，处理器核可以按存储器映射地址访问与链路口对应的 IOP 寄存器。当 LBUF 的接收缓冲不空或发送缓冲不满时，产生一个可屏蔽中断，这一中断锁存在 IRPTL 寄存器的相应位中，并可由 IMASK 寄存器的相应位屏蔽。中断服务程序在对 LBUF 读/写后应检查其空/满状态，然后决定是否返回，这样可以减少中断服务程序的调用次数。

• 链路中断服务请求 LSRQ(见表 2.84)是当链路口未指定或指定的 LBUF 被禁止时，外部设备试图访问某个链路口而引起的中断。ADSP21160 根据 LSRQ 寄存器来判断是哪个链路口被何种形式访问。当 LxACK 或 LxCLK 被外部置为有效时，就会产生一个链路服务请求 LSR，但一个处于“自环”模式的链路口不会产生 LSR。每个 LSR 在被锁存到 LSRQ 寄存器之前先要经过屏蔽“过滤”，然后 6 个可能的发送 LSR 和 6 个可能的接收 LSR“相或”，产生链路服务中断请求 LSRQ。LSRQ 能否得到响应还取决于 IMASK 寄存器的 LSRQI 位。中断服务程序必须读 LSRQ 寄存器来判断是哪个链路口的何种服务请求，方法是将 LSRQ 读入寄存器 Rx 中，用检测左起 0 值位个数的指令 Rn=LEFTZ Rx 来确定哪个链路口有服务请求(按优先级)。

5．串口

ADSP21160 有两个独立的同步串口 SPORT0、SPORT1，每个串口有自己的一套控制寄存器和数据缓冲，其用法与 ADSP2106x 的基本一致，但发送时钟频率和接收时钟频率有所不同。

串口发送时钟频率 f_{TCLK}、接收时钟频率 f_{RCLK}、ADSP21160 的核时钟频率 f_{CCLK}、时钟分频数 TCLKDIV 和 RCLKDIV 的关系为(仅对于由内部产生的串口发送、接收时钟频率，即 ICLK＝1)：

$$f_{RCLK} = \frac{f_{CCLK}}{2(RCLKDIV+1)}，\ f_{TCLK} = \frac{f_{CCLK}}{2(TCLKDIV+1)}$$

因此，ADSP21160 的串口时钟频率最大为 1/2 核时钟频率(对应 xCLKDIV=0)，具有比 ADSP2106x 更高的串口最大时钟频率。

2.4.6 ADSP2116x DSP 的汇编指令

1．ADSP2106x 的汇编代码向 ADSP2116x 移植

ADSP2116x 保持了其代码与 ADSP2106x 的高度兼容。ADSP2116x 之所以具有比 ADSP2106x 更高的性能，就是因为它具有更高的时钟频率和改进的核处理结构。例如，100 MHz 主频的 ADSP21161，仅主频速度就相当于 ADSP2106x 的 2.5 倍，因此直接把 ADSP2106x 的代码移植到 ADSP21161 上就可以获得很高的性能。但移植前需要作适当的修改，包括少数的控制、状态寄存器映射地址的变化，寄存器内部控制位和状态位的位置变化等。此外，由于 ADSP2116x 的存储器映射地址与 ADSP2106x 的不同，因此存储器空间分配也要发生相应的变化。如果要充分利用 ADSP2116x 的双运算核结构，就需要用到 ADSP2116x 的 SIMD 模式。置位 MODE1 寄存器的 PEYEN 位，就启动了 SIMD 模式(即 PEy 核使能)。在 SIMD 模式下，PEx 和 PEy 执行相同的操作，但要求被 PEx 和 PEy 处理的数据在存储器中交错放置。

ADSP2116x 增加了另一套寄存器，它们在 SIMD 模式下被隐含操作。在 SIMD 模式下，它们之间的对应关系如表 2.114 所示。

表 2.114　SIMD 模式下隐含的寄存器(Ureg－Cureg)

寄存器类型	SIMD 模式下的对应关系	
数据寄存器 dreg (或通用寄存器 ureg)	Rx－Sx (Fx－SFx)，　x=0～15	
系统寄存器 sreg(或通用寄存器 ureg)	USTAT1－USTAT2	USTAT3－USTAT4
	ASTATx－ASTATy	STKYx－STKYy
总结交换寄存器	PX1－PX2	

注意：那些不在此表内有对应关系的寄存器，在 SIMD 模式下没有隐含的对应寄存器。

2．ADSP2116x 的指令形式说明

ADSP2116x 与 ADSP2106x 的指令助记符及指令形式类似，本书不再重复介绍。读者可参阅 2.3.6 节的 ADSP2106x 指令介绍。本节只对 2.3.6 节表 2.85 中的各种指令形式在 ADSP2116x 的 SISD 模式和 SIMD 模式下的执行情况作一说明。

1) 类型 1

后修改地址操作(M 寄存器或立即数放在 I 寄存器的后面)，如 DM(Ia,Mb)=dreg 表示先将 dreg 的内容写到 Ia 指定的 DM 区的地址上，后更新 Ia，即 Ia+Mb→Ia。

预修改地址操作(M 寄存器或立即数放在 I 寄存器的前面)，如 DM(<data6>，Ia)=dreg 表示将 dreg 的内容写到 Ia+<data6>指定的 DM 区的地址上，不更新 Ia，即 Ia 的内容保持不变。后文再提到的后修改地址操作或预修改地址操作与此相同，不再解释。

- SISD 模式：数据寄存器与存储器 DM 和 PM 之间并行传递数据，计算是可选项。只支持 M 寄存器后修改地址操作。
- SIMD 模式：PEx 和 PEy 同时执行上面的 SISD 模式下的操作。

PEx 对应寄存器 I 指定的存储器地址，而 PEy 对应的存储器地址是 PEx 对应的存储器地址的下一个地址。

如果广播读位(MODE1 的 BDCST1(对应 I1)和 BDCST9(对应 I9))置位，则 PEy 与 PEx 对应同样的存储器地址，而不使 PEy 对应的存储器地址加 1。

PEx 用指定的 dreg 寄存器，而 PEy 用对应的辅助寄存器 cdreg，如表 2.114 所示。

例　R7＝BSET R6 BY R0，DM(I0，M3)=R5，PM(I11，M15)=R4；

当 ADSP2116x 工作在 SISD 模式下，上面的指令执行一条运算和两个存储器写。I0 指向 DM 区，I11 指向 PM 区。

当 ADSP2116x 工作在 SIMD 模式下，PEx 和 PEy 都同时执行一个运算和两个存储器写。PEx 中的 R7 存放运算(BSET R6 BY R0)的结果，而 PEy 中的 S7 存放运算(BSET S6 BY S0)的结果。同时，PEx 中 R5 的内容写到 I0 指向的 DM 区存储器单元中，R4 的内容写到 I11 指向的 PM 区存储器单元中；而 PEy 中 S5 的内容写到 I0+1 指向的 DM 区存储器单元中，S4 的内容写到 I11+1 指向的 PM 区存储器单元中。

例　R0＝DM(I1，M1)；

当 ADSP2116x 的广播位(MODE1 的 BDCST1)置位时，PEx 中的 R0 从 I1 指向的 DM 区存储器单元中读入数据，同时 PEy 中的 S0 也从 I1 指向的 DM 区存储器单元中读入同样的数据。

2) 类型 2

计算操作，条件是可选项。

• SISD 模式：计算或条件计算。如果是条件计算，当指定的条件满足时才执行计算操作。

• SIMD 模式：在 SIMD 模式下，PEx 和 PEy 同时执行上面的 SISD 模式下的操作。如果 PEx 中的条件满足，则 PEx 中的计算执行。如果 PEy 中的条件满足，则 PEy 中的计算执行。PEx 和 PEy 的计算执行是互相独立的，彼此不受对方条件结果的影响，只受自身条件结果的影响。

例　IF SZ R7=BSET R6 BY R0；

当 ADSP2116x 在 SISD 模式下，对 PEx 中的 ASTATx 进行条件判断，如果满足，则计算执行，结果放在 R7 中。

当 ADSP2116x 在 SIMD 模式下，分别对 PEx 中的 ASTATx 和 PEy 中的 ASTATy 进行判断。如果 PEx 中的条件满足，则 PEx 的计算(BSET R6 BY R0)执行，结果放在 R7 中；如果 PEy 中的条件满足，则 PEy 的计算(BSET S6 BY S0)执行，结果放在 S7 中；如果两者的条件都满足，两者都执行；如果两者的条件都不满足，两者都不执行。

3) 类型 3

通用寄存器与存储器 DM 区或 PM 区之间传递数据，计算和条件都是可选项。

• SISD 模式：DM 区或 PM 区存储器与通用寄存器之间传递数据，条件和计算可选。支持后修改地址操作(I 在 M 的前面)和预修改地址操作(I 在 M 的后面)。可选的后缀(LW)指定了长字地址空间(参看 2.4.3 节 ADSP21160 的存储器组织)。

• SIMD 模式：在 SIMD 模式下，PEx 和 PEy 同时执行上面的 SISD 模式下的操作。PEy 对应的存储器地址是 PEx 对应的存储器地址的下一地址，即对应 I+1(在 I 修改之前)。如果广播读位(MODE1 的 BDCST1(对应 I1)和 BDCST9(对应 I9))置位，则 PEy 与 PEx 对应同样的存储器地址，而不使地址加 1。

对通用寄存器，PEx 用指定的 ureg 而 PEy 用对应的辅助通用寄存器 cureg。如果带条件判断，PEx 和 PEy 分别对条件进行判断。如果 PEx 中条件为真，那么 PEx 就执行整条指令；如果 PEy 中的条件为真，那么 PEy 就执行整条指令。PEx 与 PEy 的条件判定互不影响。

例 R6=R3-R11，DM(I0，M1)=ASTATx；

R4=R3*R5(SSI)，PM(I9，M9)=MODE1；

第一条指令在 SISD 模式下，执行一个计算，结果放在 R6 中，同时把寄存器 ASTATx 的内容写到 I0 指向的存储器单元中。在 SIMD 模式下，PEx 和 PEy 同时执行运算，结果分别放在 R6 和 S6 中，同时 PEx 把 ASTATx 的内容写到 I0 指向的存储器单元中，而 PEy 把 ASTATy 的内容写到 I0＋1 指向的存储器单元中。

第二条指令与第一条指令类似，但对于存储器写操作，由于 MODE1 在 PEy 中没有对应的辅助寄存器(如表 2.114 所示)，所以在 SIMD 模式下，没有隐含的存储器写操作。但对于计算操作，PEx 和 PEy 都分别进行计算(即隐含 PEy 计算操作)。

4) 类型 4

数据寄存器与存储器 DM 或 PM 区之间传递数据，用 6 bit 立即数修改地址。计算和条件都是可选项。

• SISD 模式：DM 区或 PM 区存储器与数据寄存器之间传递数据，支持立即数后修改地址操作(I 在<data6>的前面)和预修改地址操作(I 在<data6>的后面)。

• SIMD 模式：PEx 和 PEy 同时执行上面的 SISD 模式下的操作。PEy 对应的存储器地址是 PEx 对应的存储器地址的下一地址，即对应 I+1(在 I 修改之前)。如果广播读位(MODE1 的 BDCST1(对应 I1)和 BDCST9(对应 I9))置位，则 PEy 与 PEx 对应同样的存储器地址，而不使地址加 1。

对数据寄存器，PEx 用指定的 dreg 而 PEy 用对应的辅助数据寄存器 cdreg。如果带条件，PEx 和 PEy 分别对条件进行判断。如果 PEx 中的条件为真，那么 PEx 就执行整条指令；如果 PEy 中的条件为真，那么 PEy 就执行整条指令。PEx 与 PEy 的条件判定互不影响。

5) 类型 5

两个通用寄存器之间传递数据，或 PEx 与 PEy 的数据寄存器内容进行交换，条件和计算为可选项。

• SISD 模式：两个通用寄存器之间传递数据(＝)，或 PEx 与 PEy 的数据寄存器内容进行交换(<－>)。如果带计算，它与寄存器之间的数据传递并行执行。如果带条件，则条件满足才执行整条指令。

• SIMD 模式：PEx 和 PEy 同时执行上面的 SISD 模式下的操作。

对于通用寄存器之间的数据传送，PEx 对应的是 uregx 和 uregy，而 PEy 对应的是 curegx 和 curegy。

表 2.115 (a)和表 2.115 (b)分别列出了 SISD 模式和 SIMD 模式下，通用寄存器之间的数据传送。

如果带条件，PEx 和 PEy 分别对条件进行判断。如果 PEx 中条件为真，那么 PEx 就执行整条指令；如果 PEy 中的条件为真，那么 PEy 就执行整条指令。PEx 与 PEy 的条件判定互不影响。

表 2.115(a)　SISD 模式下通用寄存器之间的数据传送

指　　令	显性(PEx)的数据传送	隐含(PEy)的数据传送
IF COND compute，uregx=uregy	uregy→uregx	无
IF COND compute，uregx=curegy	curegy→uregx	无
IF COND compute，curegx=uregy	uregy→curegx	无
IF COND compute，curegx=curegy	curegy→curegy	无

表 2.115(b)　SIMD 模式下通用寄存器之间的数据传送

指　　令	显性(PEx)的数据传送	隐含(PEy)的数据传送
IF COND compute，uregx=uregy	uregy→uregx	curegy→curegx
IF COND compute，uregx=curegy	curegy→uregx	uregy→curegx
IF COND compute，curegx=uregy	uregy→curegx	curegy→uregx
IF COND compute，curegx=curegy	curegy→curegy	uregy→uregx

6) 类型 6

移位器立即数操作。条件、数据寄存器与存储器之间传递数据都为可选项。

• SISD 模式：移位器立即数操作，Y 操作数是一个 8 位的立即数，或是两个 6 位的立即数。如果指定数据寄存器和存储器之间数据传递，则它与移位器操作并行执行。它只支持后修改地址操作。

• SIMD 模式：PEx 和 PEy 同时执行上面的 SISD 模式下的操作。如果指定寄存器和存储器之间数据传递，则 PEy 对应的存储器地址是 PEx 对应的存储器地址的下一个地址。如果广播读位(MODE1 的 BDCST1(对应 I1)和 BDCST9(对应 I9))置位，则 PEx 与 PEy 对应同样的存储器地址，而不使地址加 1。如果带条件，PEx 和 PEy 分别对条件进行判断；如果 PEx 中条件为真，那么 PEx 就执行整条指令；如果 PEy 中的条件为真，那么 PEy 就执行整条指令。PEx 与 PEy 的条件判定互不影响。

7) 类型 7

地址寄存器修改，条件和计算操作为可选项。

• SISD 模式：地址寄存器 I 由寄存器 M 进行修改，即 I+M→I。如果计算指定，则它与地址寄存器 I 的修改并行执行。如果条件指定，则它影响整条指令。

• SIMD 模式：与 SISD 模式相同，地址寄存器 I 由寄存器 M 进行修改，即 I+M→I。如果指定计算，则 PEx 和 PEy 同时进行计算。如果指定条件，则它影响整条指令。PEx 和 PEy 分别对条件进行判断，如果 PEx 中条件为真，那么 PEx 就执行整条指令；如果 PEy 中的条件为真，那么 PEy 就执行整条指令，即地址寄存器的修改依靠 PEx 和 PEy 两者条件判定结果的“或”。

8) 类型 8

直接地址(或 PC 相对地址)跳转/函数调用。条件为可选项。

• SISD 模式：跳转/函数调用到指定的地址或 PC 相对地址。指令附带下列的可选项：

(DB)延迟跳转，即继续执行后续两条指令后才跳转。

(LA)循环终止，迫使循环地址堆栈和 PC 堆栈弹出。如果跳转指令跳出一个循环，就要用(LA)；如果没有循环或跳转地址本身就在循环体的内部，则不要用(LA)。

(CI)清除中断。当某个中断服务程序正在执行时，(CI)清除中断标志(即中断锁存寄存器 IRPTL/LIRPTL 和中断屏蔽指针 IMASKP/LIRPTL 的相应位)以允许此中断再次被 DSP 记录(以后再响应它)，但并没有离开这个中断服务程序。指令 JUMP(CI)应该位于中断服务程序内部。

• SIMD 模式：与 SISD 模式相同，执行同样的跳转和函数调用。

如果带条件，则只有当 PEx 和 PEy 的条件都满足时，才执行跳转或函数调用，即两者判定结果相“与”。

例　IF AV JUMP(PC，0x4)(LA)；

当 ADSP2116x 在 SISD 模式下时，如果 PEx 的条件为真则执行一个 PC 相对跳转，否则不进行任何操作。

当 ADSP2116x 在 SIMD 模式下时，如果 PEx 的条件和 PEy 的条件都为真时则执行一个 PC 相对跳转，否则不执行任何操作。

9) 类型 9

间接(或 PC 相对)跳转/函数调用。条件、计算操作为可选项。

• SISD 模式：跳转/函数调用到指定的 PC 相对地址或预修改的 I 寄存器指向的地址。指令可附带(DB)、(LA)、(CI)等选项(参看指令类型 8)。

如果带条件，则当条件为真时跳转或函数调用才执行。如果是一个不带 ELSE 的计算操作，则计算操作与跳转/函数调用并行执行。如果是一个带 ELSE 的计算操作，只有当指定的条件为假时，计算操作才执行。

• SIMD 模式：与 SISD 执行同样的跳转或函数调用，但对条件操作的处理有所不同。

如果带条件，只有当 PEx 和 PEy 中的条件都为真时，跳转/函数调用才执行。如果是一个不带 ELSE 的计算操作，那么当某个运算核的条件为真时，该运算核就执行计算操作，与另一个运算核的条件判定结果无关。如果是一个带 ELSE 的计算操作，那么当某个运算核的条件为假时，该运算核就执行计算操作，与另一个运算核的条件判定结果无关。

例　IF COND JUMP $\left|\begin{array}{l}\text{(Md,Ic)}\\ \text{(PC,<reladdr6>)}\end{array}\right|$, $\left|\begin{array}{l}\text{(compute)}\\ \text{(ELSE compute)}\end{array}\right|$;

上面的指令在 SISD 模式和 SIMD 模式下的执行情况分别如表 2.116(a)和表 2.116(b)所示。

表 2.116(a)　SISD 模式下的条件执行

条件测定(PEx)	ELSE	执 行 结 果
0(FALSE)	0(无)	跳转不执行，计算不执行
0(FALSE)	1(有)	跳转不执行，计算执行
1(TRUE)	0(无)	跳转执行，计算执行
1(TRUE)	1(有)	跳转执行，计算不执行

表 2.116(b)　SIMD 模式下的条件执行

条件测定		ELSE	执 行 结 果
PEx	PEy		
0	0	0	跳转不执行，PEx 计算不执行，PEy 计算不执行
0	1	0	跳转不执行，PEx 计算不执行，PEy 计算执行
1	0	0	跳转不执行，PEx 计算执行，PEy 计算不执行
1	1	0	跳转执行，PEx 计算执行，PEy 计算执行
0	0	1	跳转不执行，PEx 计算执行，PEy 计算执行
0	1	1	跳转不执行，PEx 计算执行，PEy 计算不执行
1	0	1	跳转不执行，PEx 计算不执行，PEy 计算执行
1	1	1	跳转执行，PEx 计算不执行，PEy 计算不执行

10) 类型 10

间接(或 PC 相对)跳转，同时存储器与数据寄存器之间进行数据传送。计算操作为可选项。

• SISD 模式：条件跳转到指定的 PC 相对地址或预修改的 I 寄存器指向的地址。同时此类型也提供了存储器与数据寄存器之间的数据传送，且与跳转指令并行工作。如果带计算操作，三者同时并行工作。此类型中，只有计算操作是可选的，条件与 ELSE 都是必须的。数据传送只支持后修改地址操作，不支持预修改地址操作。如果指定的条件为真，则跳转执行，如果指定的条件为假，则数据传送和计算操作(如果指定)并行执行。

此类型不可以带(DB)、(LA)、(CI)等选项。

• SIMD 模式：与 SISD 模式相同，也是条件跳转。但对条件的处理不同。该模式下 PEx 和 PEy 中的条件必须都为真，跳转才执行。哪一个运算核的条件为假，它的数据传送和计算(如果指定)就会执行，而与另一个运算核的条件判定无关。

对于计算操作，PEx 用指定的 dreg 寄存器，而 PEy 用相应的辅助数据寄存器 cdreg。

对于数据传送，PEx 对应寄存器 I 指向的存储器单元，而 PEy 对应 I+1 指向的存储器单元。寄存器 I 由寄存器 M 进行后修改地址操作，此类型不支持预修改地址操作。

11) 类型 11

函数或中断服务程序返回。条件、计算操作为可选项。

• SISD 模式：子函数返回(RTS)或中断服务程序返回(RTI)。返回使得处理器从 PC 堆栈顶的地址处开始执行。RTS 和 RTI 的区别在于 RTI 要多做两个操作：第一，如果 ASTAT 和 MODE1 寄存器已经入栈的话，则应使它们弹出栈，这适用于 IRQ0～2、定时中断和 VIPRT 中断；第二，将 IRPTL、LIRPTL 和 IMASKP 中的相应位清 0。

RTS 和 RTI 支持(DB)、(LR)后缀选项。如果返回不是延迟的，而且返回指令位于循环的最后 3 个指令内，就必须用(LR)。(LR)可以确保正确的重新进入循环。如果中断服务程序中有 JUMP(CI)指令，它的返回 RTS 必须加(LR)。

• SIMD 模式：与 SISD 模式相同，也是子函数返回(RTS)或中断服务程序返回(RTI)。但如果带条件，则对条件的处理不同。在此模式下，PEx 和 PEy 中的条件必须都为真，返回才执行。

对于计算操作，PEx 用指定的 dreg 寄存器，而 PEy 用相应的辅助数据寄存器 cdreg。

12) 类型 12

执行循环操作，直到循环计数器变为 0。

SISD 模式与 SIMD 模式的处理相同。对循环计数器 LCNTR 的置数，可以是 16 位的立即数，也可以通过一个通用寄存器来进行。循环的开始地址被压入 PC 堆栈，循环结束地址和 LCE 条件被压入到循环地址堆栈。结束地址可以是 24 位的绝对地址，也可以是 24 位的 PC 相对地址。

13) 类型 13

执行循环操作，直到某个条件为真。

SIMD 与 SISD 模式相同，但对条件的处理不同。在 SIMD 模式下，PEx 和 PEy 的条件都为真时，循环才结束。

14) 类型 14

存储器与通用寄存器之间的数据传送，直接寻址。

在 SISD 与 SIMD 模式下一样，都是存储器与通用寄存器之间的数据传送，直接寻址。但在 SIMD 模式下，PEx 和 PEy 并行进行数据传送。PEy 对应的存储器地址是 PEx 对应的存储器地址的下一个地址。对于通用寄存器，PEx 对应的是 ureg，而 PEy 对应的是 cureg。后缀选项(LW)表示指定长字(64 bit)地址空间，而默认为正常字(32 bit)地址空间。

15) 类型 15

存储器与通用寄存器之间的数据传送，用立即数预修改地址。

在 SISD 与 SIMD 模式下一样，都是存储器与通用寄存器之间的数据传送，立即数预修改地址操作。但在 SIMD 模式下，PEx 和 PEy 并行进行数据传送。PEy 对应的存储器地址是 PEx 对应的存储器地址的下一个地址。对于通用寄存器，PEx 对应的是 ureg，而 PEy 对应的是 cureg。后缀选项(LW)表示指定长字(64 bit)地址空间，而默认为正常字(32 bit)地址空间。

16) 类型 16

32 位立即数写到存储器中，用 M 寄存器后修改地址。

在 SISD 与 SIMD 模式下一样，都是把立即数写到存储器中，M 寄存器后修改地址操作。但在 SIMD 模式下，PEx 和 PEy 并行进行数据传送。PEy 对应的存储器地址是 PEx 对应的存储器地址的下一个地址。

17) 类型 17

32 位的立即数写到通用寄存器中。

在 SISD 与 SIMD 模式下一样，都是立即数写到通用寄存器中。但在 SIMD 模式下，32 位的立即数同时写到 PEx 和 PEy 的相应寄存器中。PEx 对应的寄存器为 ureg，而 PEy 对应的寄存器为 cureg。

18) 类型 18

按照<data32>中位的置位(非 0 位)对系统寄存器 sreg 的相应位进行各种操作：SET(置位)、CLR(清 0)、TGL(取反)、TST(测试)、XOR(异或)。

测试操作：当<data32>中非 0 位所对应的 sreg 中的各位都置位时，它就置位测试标志位 BTF(在 ASTATx/y 中)。

异或操作：当<data32>中各位与 sreg 中的各位都相同时，它就置位测试标志位 BTF(在

ASTATx/y 中)。

在 SISD 与 SIMD 模式下的操作相同，但在 SIMD 模式下 PEx 和 PEy 同时工作。PEx 对应的系统寄存器为 sreg，而 PEy 对应的系统寄存器为 csreg。PEx 对应的标志寄存器为 ASTATx，而 PEy 对应的标志寄存器为 ASTATy。

19) 类型 19

MODIFY：修改 DAG1 的 I0～I7 寄存器或 DAG2 的 I8～I15 寄存器。

BITREV：位反序方式的地址修改。

SISD 模式与 SIMD 模式下的操作相同。

20) 类型 20

对下述对象作压栈或出栈操作：循环地址和循环计数器(LOOP)、状态寄存器(STS)、PC 堆栈(PCSTK)、刷新(清除)指令 CACHE。这 4 个操作可以在一个周期内完成。但 PUSH 与 POP 不能同时出现。

SISD 模式与 SIMD 模式下的操作相同。

21) 类型 21

空操作，仅程序地址加 1。

SISD 模式与 SIMD 模式下的操作相同。

22) 类型 22

执行类似 NOP 的空操作，DSP 进入低功耗状态，直到某个中断出现。

SISD 模式与 SIMD 模式下的操作相同。

23) 类型 23

仅由 C 编译器产生。

CJUMP 相当于通常跳转指令操作和帧指针 I6、堆栈指针 I7 的保存操作。

SISD 模式与 SIMD 模式下的操作相同。

关于 ADSP2116x 的 C/C++语言编程问题，用户可查阅 2.3.7 节的内容。

思　考　题

2.1　本章介绍的 DSP 类型中哪些是定点 DSP，哪些是浮点 DSP？浮点 DSP 和定点 DSP 在编程时有何区别？在应用定点 DSP 指令时，如何防止溢出？

2.2　本章以对比的方式介绍了当前最为流行的几种 DSP 的内部功能结构，试总结出不同 DSP 的共同特点？

2.3　不同类型的 DSP 具有不同的应用场合，如何根据实际应用要求选取 DSP 类型？

2.4　什么是冯·诺依曼结构、流水线结构、超标量结构？这些 DSP 分别采用哪些结构和技术来提高其性能？

2.5　TMS320C5000 DSP 和 TMS320C6000 DSP 都是 TI 公司推出的当前比较流行的 DSP，它们的主要不同点和相同点在哪里？它们分别主要应用于哪些领域？

2.6　TMS320C6000 DSP 的片内具有 8 个并行功能单元，考虑如何充分发挥这 8 个功能单元的并行能力？

2.7　AD 公司推出的 SHARC DSP 包括 ADSP2106x 和 ADSP2116x，它们的主要不同点和相同点在哪里？ADSP2116x DSP 在哪些方面比 ADSP2106x DSP 有所提高？

2.8　DSP 有哪些中断？什么是中断矢量表？其地址空间在何处以及如何修改？DSP 响应中断的条件是什么？DSP 响应中断时，会自动进行哪些操作？

2.9　什么是存储器映射寄存器？访问这些寄存器时有何特殊要求？

2.10　DSP 的片内存储器如何配置？哪些可以作为程序存储区、数据存储区、程序 CACHE 和数据 CACHE？CACHE 有何用途？

2.11　DSP 对片内存储器的访问比对片外存储器的访问(零等待)快多少？为什么把主要的运算程序和数据都放在片内存储器上？

2.12　DMA 有何用途？如何利用 DMA 来提高 DSP 的执行速度？如何编写 DMA 程序？

2.13　不同 DSP 的片内外设有哪些？如何应用 SHARC DSP 的链路口？

2.14　何为嵌入式实时系统？JTAG 标准是什么？

2.15　汇编程序中的伪指令有何作用？其与真正的处理器命令有何不同？

2.16　不同 DSP 的每一指令周期内最多可以完成多少条指令？是否主频高就意味着运算速度快？比较这些 DSP 的 1024 点 FFT 的执行时间。

2.17　ADSP2116x 的 SIMD 模式如何启动？应用 SIMD 模式时，对数据存放有何要求？ADSP21161 是一种低功耗、低成本产品，具有 1 MB 的片内存储器，问 4096 点复数 FFT 是否可以应用 SIMD 模式？8192 点复数 FFT 呢？

2.18　不同 DSP 支持的寻址方式有哪些？什么是循环寻址、位反序寻址，哪些 DSP 支持这种寻址方式？

2.19　TMS320C5000/C6000 DSP 的开发工具 CCS 中用到链接命令文件(.cmd)，而 SHARC DSP 的开发工具 VisualDSP++中用到链接描述文件(.ldf)，这两种文件的用途是什么？它们的用途是否相同？

2.20　什么是堆栈？C/C++编译器为什么会用到堆栈？如何设置堆栈大小？如果堆栈设置太小，程序执行过程中会不会发生问题，如何克服？是否所有 DSP 都有专门的堆栈指针？

2.21　标准 C/C++语言中定义的数据类型对于每种 DSP 的 C/C++编译器是否一样？C/C++语言和汇编语言混合编程时要注意的原则是什么？为什么要注意？C/C++编译器会自动输出哪些段？用到哪些寄存器？如何深刻理解 C/C++编译器环境，才能在混合编程时不容易出现问题？

第 3 章　TMS320C5000/C6000 DSP 集成开发环境 CCS IDE

本章介绍 TI 的 CCSv2.0 集成开发环境。CCS 几乎对 TI DSP 软件开发的整个过程提供支持，包括对 C/C++和汇编代码编辑、可执行代码生成、语法和逻辑错误调试、实时调试和测试的支持。如果把 MATLAB 和 CCS 相结合，就可以在统一的 MATLAB 环境下完成从概念设计到实时实现的整个过程。

3.1　TI CCS 概述

3.1.1　CCS 的特点及功能概述

为了缩短 TI DSP 的软件开发周期，TI 公司针对 TMS320C2000、TMS320C5000 和 C6000 各提供了一套集成开发环境 CCS IDE(Code Composer Studio Integrated Development Environment)。它们的功能和操作方法非常类似。本章以 CCS6000(以下简称 CCS)的 2.0 版本(即 CCSv2.0)为例，详细介绍其功能及操作方法。

CCS 是一个开放式和具有强大集成能力的开发环境，该开发环境集工程文件管理工具、代码编辑工具、代码生成工具(Build)、代码调试工具、代码实时调试和测试工具(DSP/BIOS 和 RTDX)以及其它一些工具和插件为一体，几乎能完成 DSP 软件开发过程的各个环节。同时，它又具有可供第三方接入的开放式结构，这种开放式结构使得开发人员可以采用特定的工具自定义环境，以满足特殊的设计需要。CCS 的组件结构如图 3.1 所示。

CCS 提供了非常良好的用户界面，面向窗口，具有菜单、对话框式接口，具备丰富的图形图标，辅之以完整的可即时访问的在线帮助，使开发人员不必记忆复杂命令，就能够轻松地掌握和使用 CCS 开发系统。

1．工程管理工具

CCS 对一个 DSP 应用系统的文件管理是通过工程方式进行的。对于熟悉 VC 程序开发的人员来说，工程方式管理是不难理解的。工程中包含着系统所有的源代码文件、目标代码文件、链接命令文件、配置文件、库函数、头文件等。采用工程的方式统一管理各种文件，非常直观、灵活，这是 CCS 不同于传统开发工具的一大改进之处。

- 向工程中添加文件，CCS 会根据文件的类型将其自动分配到相应的文件夹中。
- 不用添加头文件，CCS 会自动搜索源文件中用到的头文件，并添加到工程中。

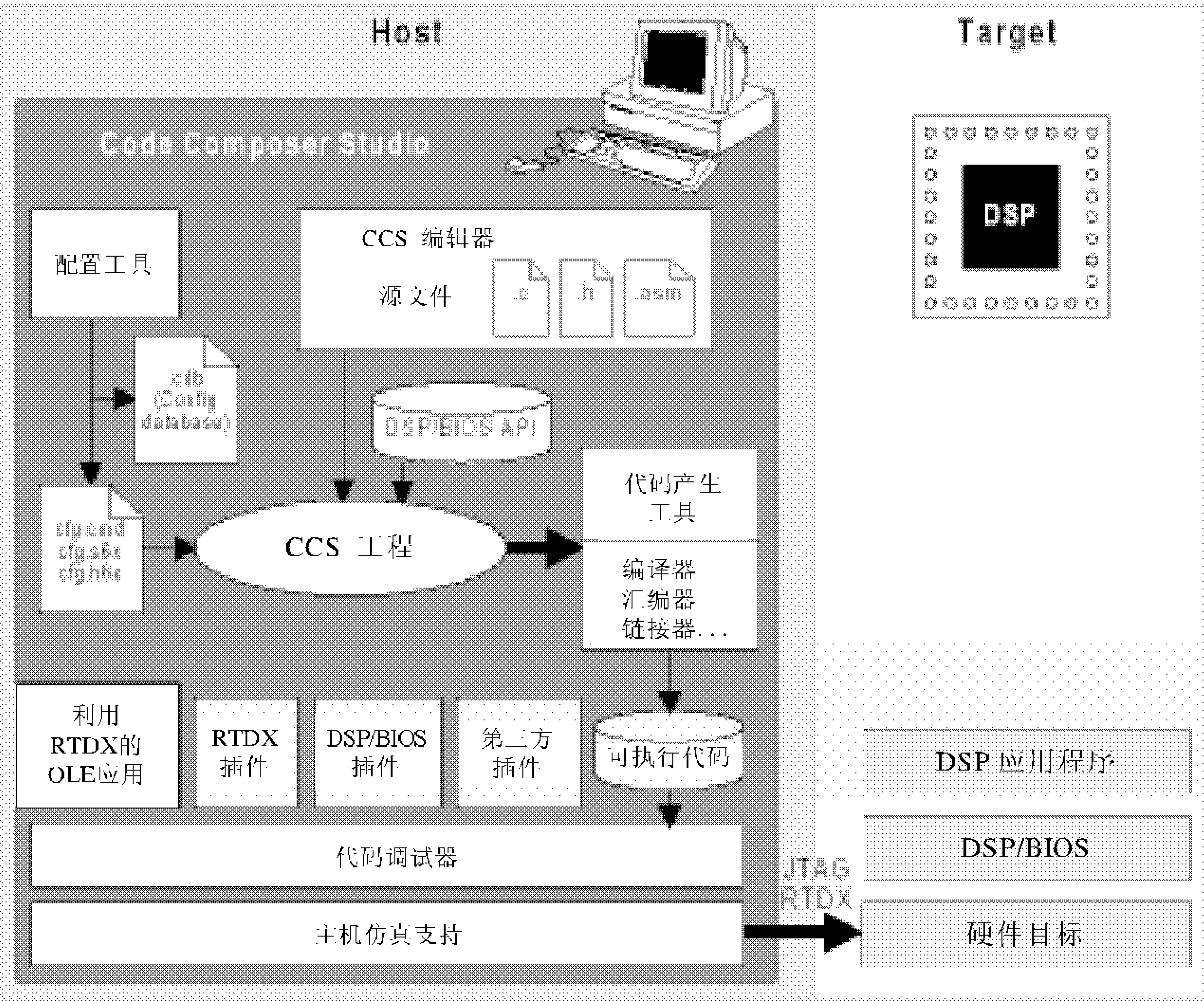


图3.1 CCS的组件结构

2．源代码编辑工具

• 对C语言和汇编语言的源代码进行编辑，同时设计者可以采用C语言和汇编语言的混合显示模式，即在每一条C指令后显示出相应的汇编语言指令。

• 对关键字、注释、字符串等以不同的颜色高亮显示。

• 能够在一个或多个文件中查找、替换、快速搜寻特定字符串。同时，CCS编辑器下的一些常用命令如文件的生成、打开、存储以及文本的剪切、拷贝、粘贴、列编辑等和常用编辑软件一样，便于掌握。

• 高亮选定某一指令(C语言或汇编)后，按下F1键，可以得到此指令的帮助。

• 提供上下文相关的帮助。

• 用户可以根据自己的习惯定制不同的快捷方式。

3．代码产生工具

在CCS开发环境下，每一个DSP应用系统所用到的源代码文件、目标文件、库文件、链接命令文件、配置文件等都包含在相应的工程中。CCS对某一应用系统的代码生成(Build)，实际上就是对这一工程文件的编译、汇编和链接过程，它所实现的功能和传统DSP开发系统中的makefile命令功能相同，而且CCS也支持传统的makefile命令输入。

• 通过对话窗方式设置Build命令选项。

• 对整个工程编译和链接，或仅对修改后的内容进行编译和链接，或只对某一文件编译。

• 扫描文件，为整个工程创建依赖关系树。

• 自动错误定位。源代码编译、汇编后，可以给出错误、警告信息，双击某个错误，CCS即可自动打开有错误的源文件，并且光标会停在出错行。

4．代码调试工具

CCS 支持如下代码调试功能：

- 提供完善的程序运行控制功能，如条件执行、单步执行、断点设置和清除等。
- 在断点(breakpoint)处自动更新所有窗口。
- 显示和修改变量、数组、结构体等数据类型的内容。
- 显示和修改存储器、寄存器的内容。
- 观察堆栈使用情况。
- 使用探点(probepoint)工具将数据从主机的文件传到 DSP 的存储器中，或将数据从 DSP 的存储器输出到主机的某一文件中。
- 图形显示 DSP 中的数据(静态或动态显示)。
- 统计代码的执行性能。
- 显示反汇编代码和 C 代码，实现 C 和汇编代码的同时调试。
- 提供 GEL 命令输入框，开发人员可以在 CCS 菜单中添加经常用到的功能。

5．实时调试和测试工具

传统调试工具对与时间细节有关的问题的分析能力不足，而 CCS 的 DSP/BIOS 插件提供了这种实时分析功能，这是 CCS 不同于传统调试工具的一大特色之一。利用 DSP/BIOS 插件，可以在对实时性能影响最小的情况下(即使 DSP/BIOS 在全功能下工作，也占不到 1 K 字的程序存储空间和不到 1 个 MIPS 的 DSP 运行时间)，完成以下实时分析功能：

- DSP 应用程序的实时跟踪：实时跟踪程序的动态过程并把信息记录下来，再利用 CCS 中的分析工具进行动态显示。
- 代码性能统计：利用统计(statistics)功能块等监视代码对 DSP 资源的使用情况。
- 文件数据流控制：目标程序在执行过程中可以对主机文件进行访问，而不用使运行的程序停下来。

与传统的调试工具不同，DSP/BIOS 提供的这些实时分析功能是通过 DSP 应用程序(C 语言程序或汇编程序)调用相应的 DSP/BIOS API 功能模块实现的，经过编译和链接后，这些 DSP/BIOS 功能模块与目标程序一起生成可执行文件并映射到 DSP 的存储空间中。这种实时分析功能是通过 DSP/BIOS 功能模块在目标 DSP 中与应用程序一起运行获得的，这些 DSP/BIOS 功能模块占用极少的 DSP 资源，对应用程序的实时性影响非常小，因此可以获得应用程序真实的实时性信息。而传统的调试工具是主机通过 JTAG 仿真器非实时地扫描 DSP 资源来获得程序执行信息的，因此实时性不够。

CCS 还提供了 DSP/BIOS 配置工具和可视化实时分析工具，用来方便地配置目标程序中用到的 DSP/BIOS 功能模块以及显示这些功能模块获取的应用程序的实时性信息。

这些 DSP/BIOS 功能模块不但可用于应用程序的实时调试阶段，甚至可以作为应用程序的 I/O 部分，用于从主机获取原始数据和向主机发送处理结果，因此可以大大节省整个软件系统的开发周期。

3.1.2　CCS 支持的调试器

用 C 或汇编语言开发的应用程序，经代码生成工具(Build)编译、链接生成可执行文件

(*.out)后，还需要把可执行文件加载到指定的 DSP 目标板或软件模拟器中进行逻辑错误和实时性调试。CCS(CCS6000)支持的调试器包括：Simulator(软件模拟器)、C6x01 EVM 板、C6x11 DSK 板和硬件仿真器 XDS510。

1．Simulator

Simulator 是运行在主机上的一个软件程序包，它利用主机的处理器和存储器资源对 TI DSP 的指令运行进行模拟。Simulator 在没有硬件目标板的条件下实现对 TI DSP 应用程序的调试。

Simulator 几乎可以模拟除 DSP/BIOS 部分功能外的所有目标 DSP 执行情况，它包括如下特点：

• 提供程序运行控制，如单步执行、条件执行等。

• 设置断点(Breakpoint)和探点(Probepoint)。

• 可以显示和修改变量、数组、结构体等符号内容以及存储器和寄存器中的值。

• 统计代码的执行性能(Profiler)。

• 模拟 cache、流水定时、外部中断、I/O 设备等，甚至可以模拟利用 RTDX 进行数据传递。

2．C6x01 EVM 板

TI 公司提供的 C6x01 EVM 目标板，为 TI C6000 DSP 应用程序的开发者提供了一个硬件平台，它与 CCS 集成开发环境相结合可以实现整个应用软件开发过程(代码编写、可执行代码生成、逻辑错误调试和实时性调试)。

C6x01 EVM 目标板是主机的 PCI 插件板，可以插入主机的 PCI 插槽中，不需要硬件仿真器就可以进行程序加载和调试。C6x01 EVM 也可以不用插入主机的 PCI 插槽中，通过外接电源，利用硬件仿真器(XDS510)来加载和调试程序。C6x01 EVM 板的主要特点包括：

• 具有 PCI 接口，并且从 PCI 总线可以访问 DSP 的所有存储器资源。

• 具有一个 C6201(定点)或 C6701(浮点)DSP，并且有 4 种晶振频率选择(C6201 EVM：33.25 MHz、133 MHz、50 MHz、200 MHz，C6701 EVM：25 MHz、100 MHz、33.25 MHz、133 MHz)。

• 板上具有 64 K×32 bit SBSRAM 和 2 M×32 bit SDRAM 外部存储器。

• 具有 JTAG 仿真接口(与 PCI 和硬件仿真器连接)和 JTAG 插头(与 XDS510 硬件仿真器相连)。

• 具有 DIP 开关，用来设置 EVM 板的模式，也可以用来控制应用程序的运行过程。

• 具有一个内部发光二极管(LED1)和一个外部发光二极管(LED0)，一个用来指示应用程序的运行状态，另外一个用作加电指示灯。

• 具有立体(左右声道)16 位音频编/解码器，提供从模拟输入到 DSP 的数字信号输出(A/D 转换)和从 DSP 的数字信号输入到模拟输出(D/A 转换)的功能，采样频率为 5.5～48 KHz。

• 具有扩展存储器接口和 DSP 外围设备接口，提供用户或第三方子板支持。

3．C6x11 DSK 板

TI 公司提供的 C6x11 DSK 板具有高性价比，不但可以用来学习 C6000 DSP 软件编程技

术，而且也可用来快速开发嵌入式实时应用产品，例如网络、通讯和图像处理等产品。

C6x11 DSK 目标板既可以通过板上的并口与主机相连，加载可执行代码和调试，还可以通过硬件仿真器 XDS510 与板上的 JTAG 插头连接，进行加载和调试。C6x11 DSK 目标板的主要特点包括：

- 具有并口控制器接口，可以与主机的并口连接，通过并口可以访问 DSP 的所有存储器资源。
- 具有一个 C6211(定点)或 C6711(浮点)DSP，并且 CPU 时钟频率固定为 150 MHz。
- 具有 16 M 字节 SDRAM(C6211 DSK 为 4 M 字节)和 128 K 字节闪存。
- 具有单通道、16 位编/解码器，提供从模拟输入到 DSP 数字信号输出(A/D 转换)和从 DSP 的数字信号输入到模拟信号输出(D/A 转换)的功能，采样频率固定为 8 KHz。
- 具有 JTAG 仿真器接口(与并口和硬件仿真器连接)和 JTAG 插头(与 XDS510 硬件仿真器相连)。
- 具有扩展子板的接口。
- 具有 DIP 开关，用户可以用来控制应用程序的运行过程。
- 具有三个用户发光二极管(User_LED1, User_LED2, User_LED3)，用来指示应用程序的运行状态，另三个发光二极管分别用来指示 DSK 板上电、JTAG TBC 的使用和复位操作。

图 3.2 为 TMS320C6711 DSK 板。

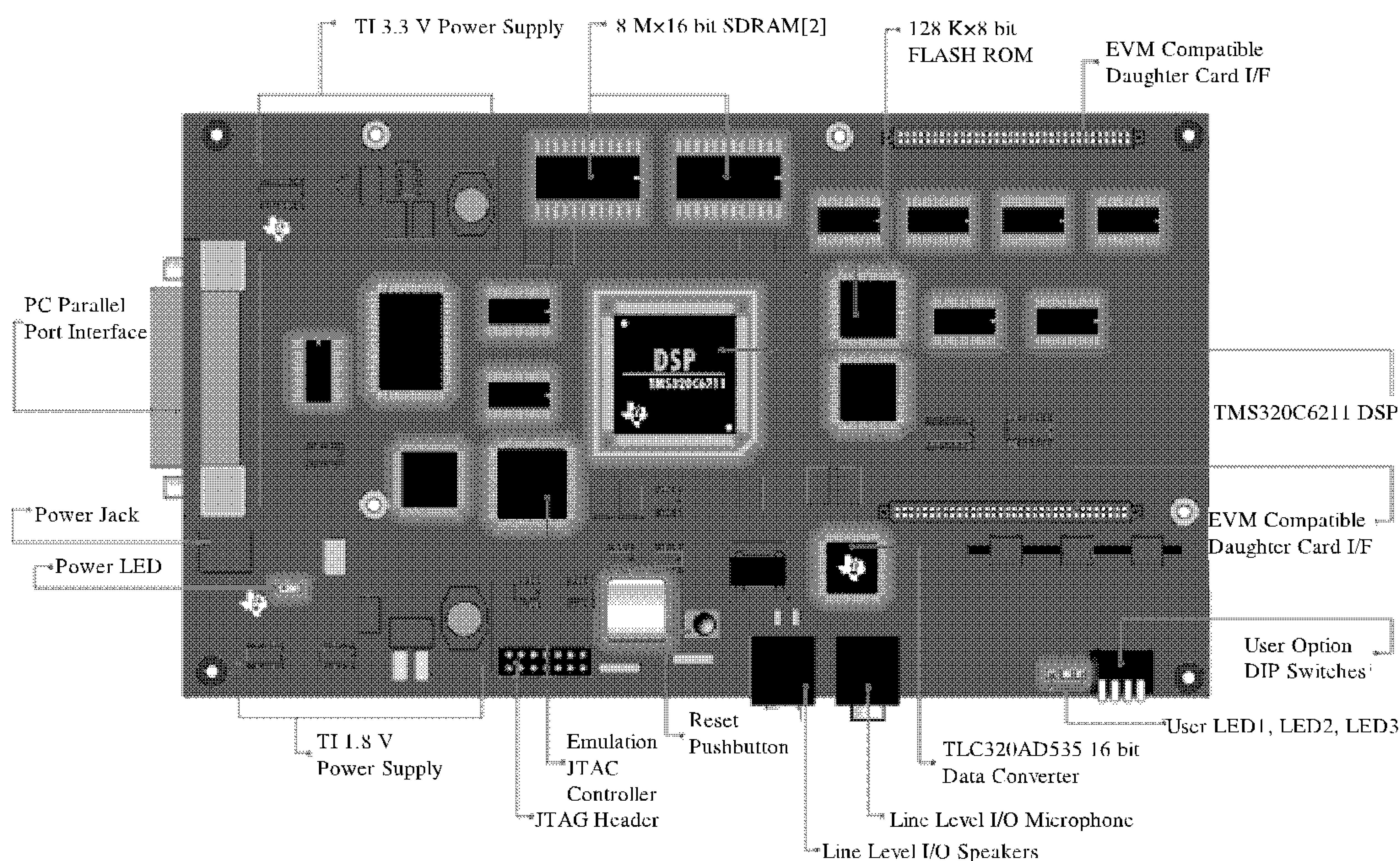


图 3.2　TMS320C6711 DSK 板

4．硬件仿真器 XDS510

CCS 支持硬件仿真器(Emulator)XDS510，用来向目标板(用户自己开发的 DSP 板或 TI 公司提供的 EVM、DSK 板)加载可执行代码和进行代码调试。在 CCS 环境下，通过硬件仿真器 XDS510 调试目标板、通过 PCI 总线调试 EVM 板、通过并口调试 DSK 板的界面、功能及操作方法与软件模拟器的完全一样(注：软件模拟器不能对代码的实时性进行调试)。

TI 公司的各种 DSP，如 C2000、C5000、C6000、C4x 的仿真器接口是兼容的(都是 JTAG 标准)，只要利用硬件仿真器 XDS510 和 CCS，就可实现对上述 DSP 的加载和调试。C3x DSP 的仿真器接口为非 JTAG 标准，使用 XDS510 时需要配转换头。

利用硬件仿真器调试目标板时，必须确保主机和目标板的地线连接可靠。在运行仿真器软件时不要插拔仿真头，最好是在退出仿真器、目标板断电后再插拔仿真头，以防止损坏仿真头或 DSP。

硬件仿真器 XDS510 的主要特点包括：

- 能够全部运行、停止和给 JTAG 链中的所有并行处理 DSP 设置断点。
- 支持高级语言调试能力。
- 能够设置软件断点，跟踪所有程序和数据寻址。
- 完善的程序运行控制。
- 加载、查看和修改所有寄存器和存储器内容。
- 以 DSP 的时钟周期作为执行时间的基准。

3.1.3 CCS 的配置与启动

安装完 CCSv2.0 后，会在桌面上出现两个图标：CCS 图标和 Setup CCS 图标。CCS 提供了对软件模拟器、硬件仿真器、EVM 板和 DSK 板的支持，因此在启动 CCS 前，用户应根据自己所拥有的软硬件资源，对 CCS 进行适当的配置。把硬件仿真器、目标板(用户自己开发的 DSP 板、DSK 板或 EVM 板)和主机连接好，并给目标板加电后，按以下步骤完成对 CCS 的配置：

(1) 双击桌面上的 Setup CCS 图标(或从 Start 菜单中选择)，打开 CCS 配置窗口，如图 3.3 所示。

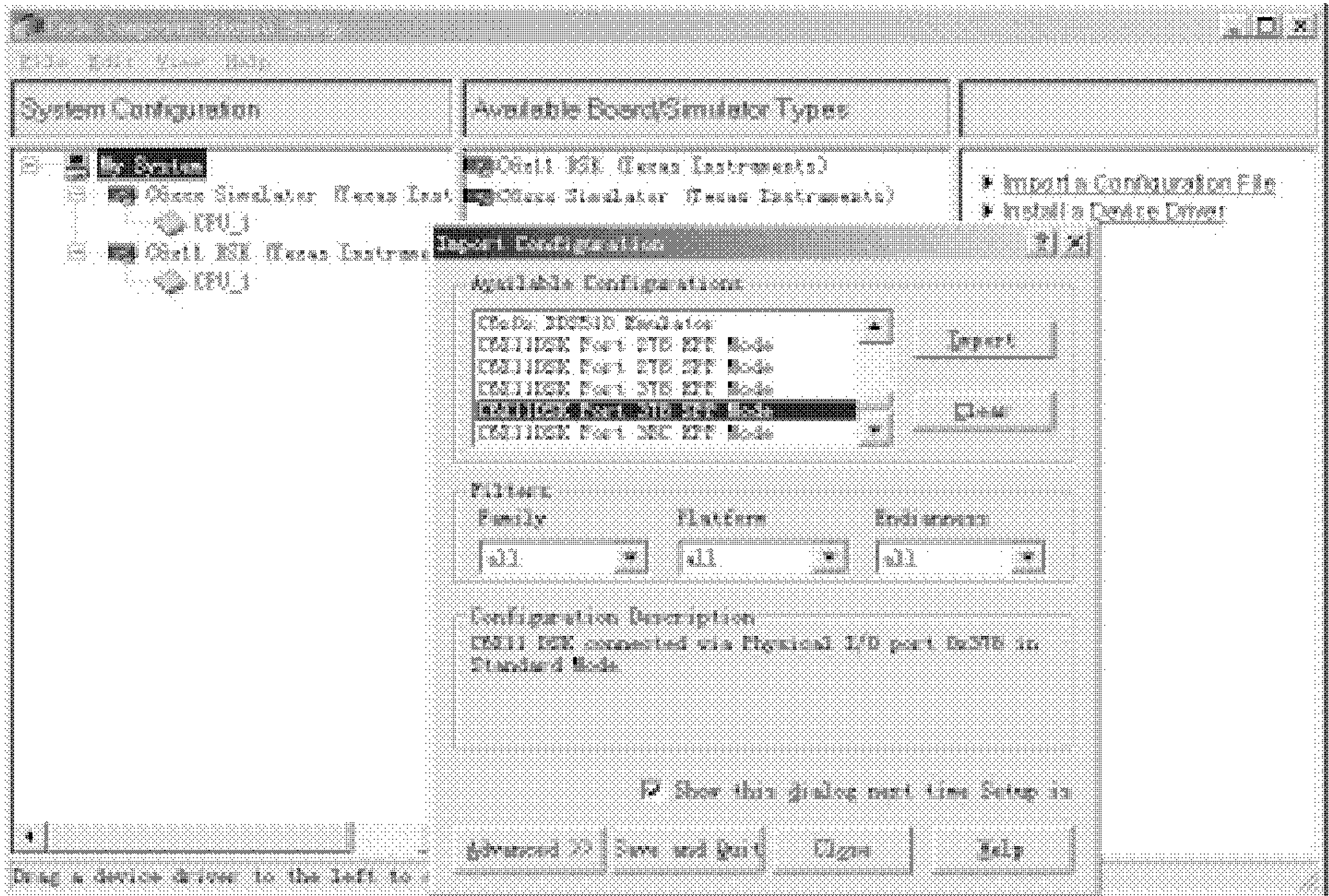


图 3.3 CCS 配置窗口

(2) 选择 File→Import，打开 Import Configuration 对话框。此对话框的 Available Configurations 列表中列出了 CCS 提供的所有标准配置。

(3) 点击 Import Configuration 对话框的 Clear 按钮，清除掉先前定义的配置文件，在弹出的确认对话框中点击 Yes，确认删除先前定义的配置文件。

(4) 在 Import Configuration 对话框的 Available Configurations 列表中选择满足用户系统的标准配置文件，在对话框的 Configuration Description 区域会显示对所选配置文件的描述。例如图 3.3 的 Import Configurations 对话框中选择了 C6X11DSK Port 378 SPP Mode 标准配置。

如果 Available Configurations 中提供的所有标准配置都不能满足用户系统要求，用户必须创建自己的配置文件。

(5) 点击 Import Configuration 对话框上的 Import 按钮，向 CCS 输入所选的系统配置。

CCS 配置窗口的 My System 图标下会自动添加所选的系统配置。

如果用户需要对多个目标板和软件模拟器进行配置，则需重复第 4 步到第 5 步直到为所有目标板和软件模拟器选择好配置文件。

(6) 点击 CCS 配置窗口的 Save and Quit 按钮，保存当前配置并退出 CCS 配置程序。

(7) 双击桌面上的 CCS 2 图标(在退出 CCS 配置程序时，系统会询问是否启动 CCS，选择 Yes 启动 CCS)，会出现一个 CCS:Parallel Debug Manager 调试工具栏。

(8) 从 Open 菜单中选择已配置好的目标板或软件模拟器，打开 CCS IDE，CCS IDE 的主界面如图 3.4 所示。

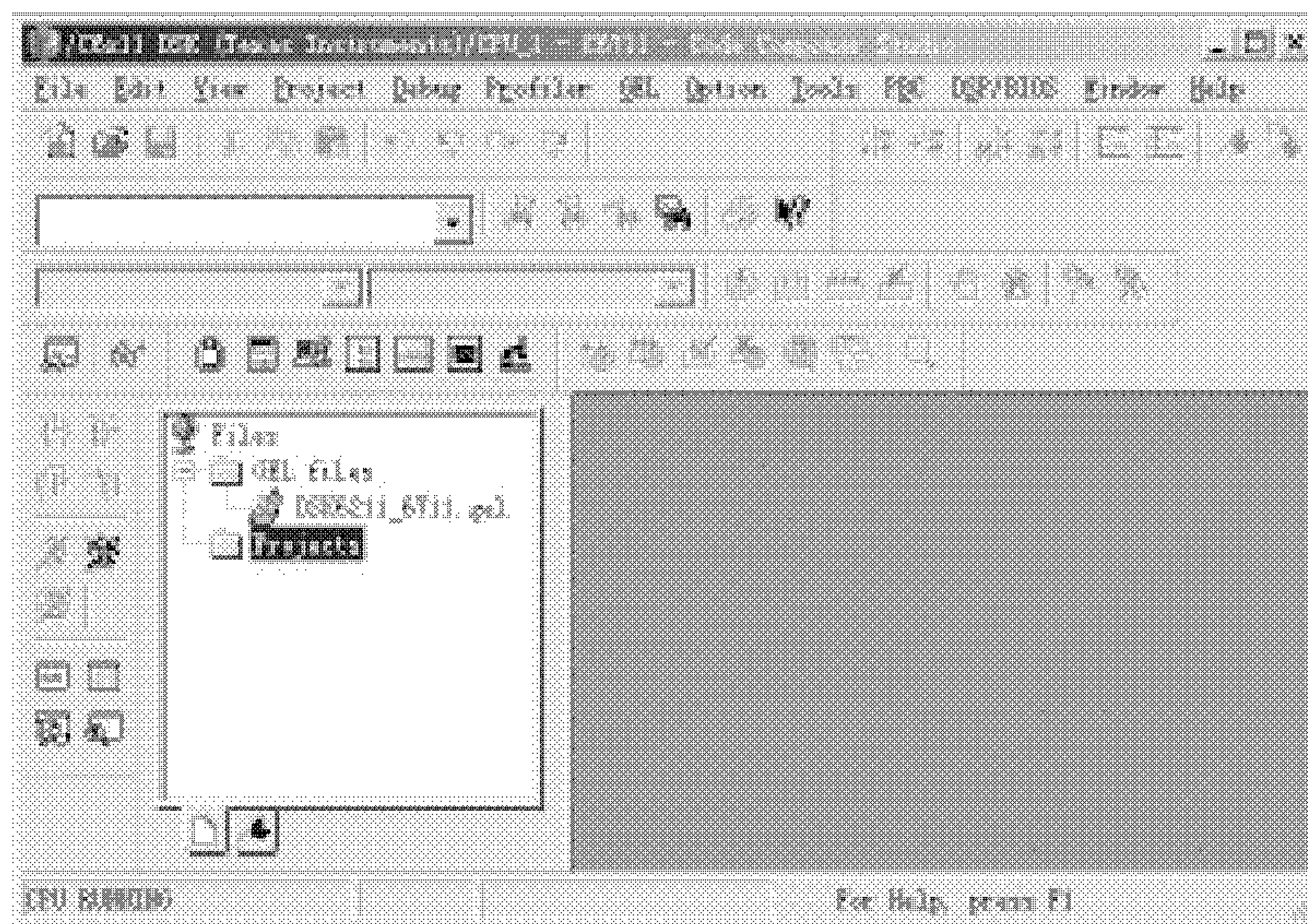


图 3.4　CCS IDE 的主界面

3.2　代码产生工具

代码产生工具 Build 是任何开发工具的基本组成部分，本节介绍在 CCS 环境下的代码产生过程和主要工具以及如何设置工程或文件的编译、链接(Build)选项。本节最后利用例子来演示从工程创建到可执行代码加载运行的整个过程。

3.2.1　代码产生过程及工具

DSP应用程序一般采用C/C++语言和汇编语言混合编程方法，即利用C/C++程序构建应用程序的主框架而用汇编语言编写关键的运算子程序。图3.5为一般DSP实时应用程序的开发流程，其中主要的开发路径如图中阴影部分所示，其它部分是可选的。

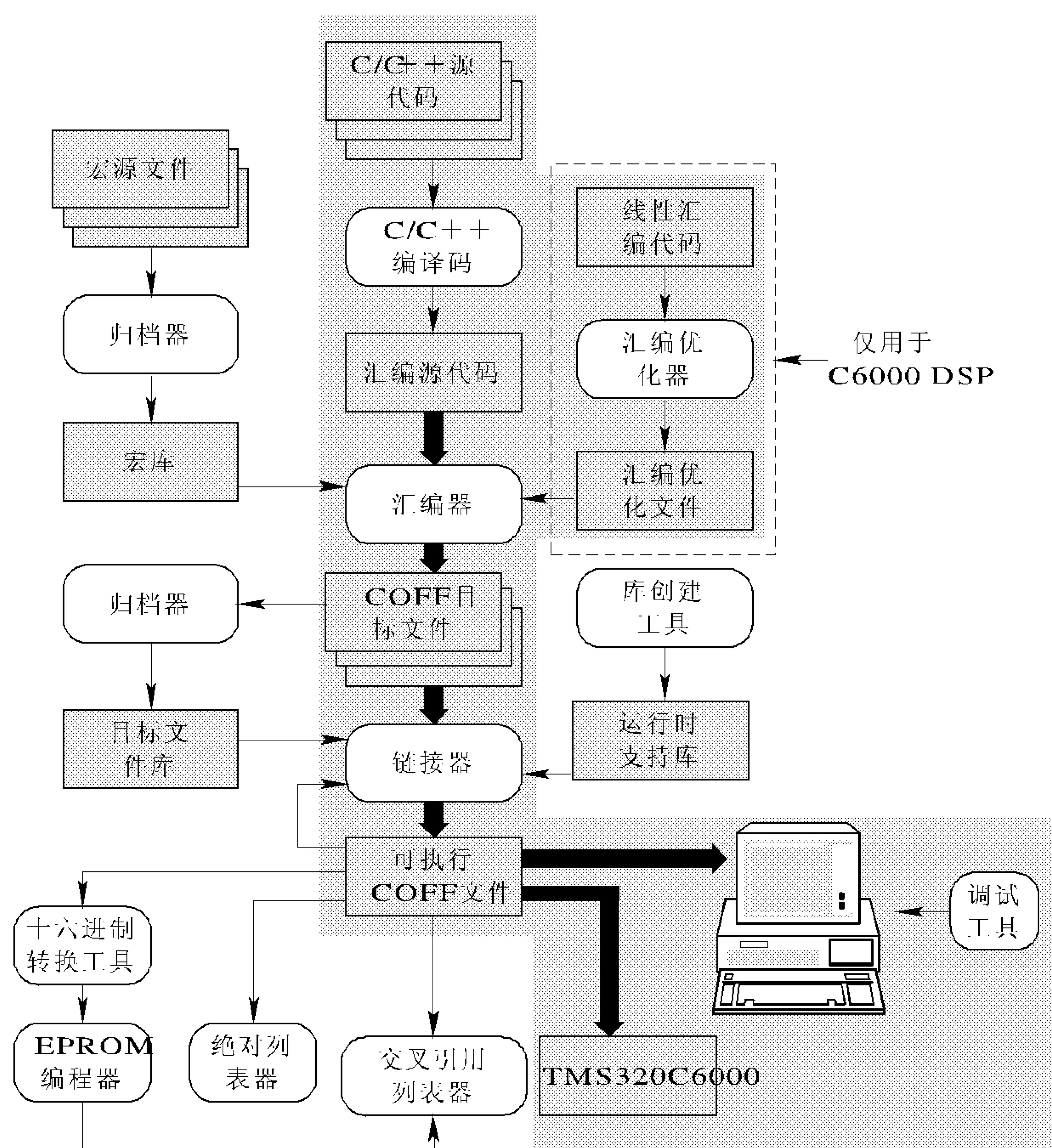


图3.5　CCS下的DSP实时应用程序开发流程

CCS的主要代码产生工具说明如下。

1．C/C++编译器(Compiler)

C/C++编译器把C/C++源程序转换成DSP的汇编程序。它由三部分组成：分析器、优化器和代码产生器。

分析器检查输入的C/C++语言程序有无语义、语法错误，然后产生一个中间文件(*.if)，它的运行包括预处理代码及代码分析两个阶段。

优化器是分析器和代码产生器之间的一个可选择的路径，其输入是分析器产生的中间文件(*.if)，优化器对其优化后，产生一个高效版本的文件(*.opt)，其优化的级别作为选项由用户来指定。

代码产生器以分析器生成的文件(*.if)和/或优化器生成的文件(*.opt)作为输入，生成一个汇编语言文件。

2．汇编器(Assembler)

汇编器把汇编语言源代码转换成 COFF 格式的机器语言代码。汇编源代码可以包含 DSP 指令、汇编器伪指令及宏指令。用户可以用汇编伪指令控制汇编过程的各个方面。汇编器具有以下功能：

- 处理文本文件中的源语句，生成可重定位的目标文件。
- 生成源程序列表，并向用户提供对此列表的控制。
- 允许用户把代码分成段，并为目标代码中的每一段都保持 SPC(段程序计数器)。
- 汇编条件段。
- 定义和引用全局符号。
- 支持宏，允许用户在线定义宏或库内定义宏。

3．链接器(Linker)

链接器把多个 COFF 目标文件链接成单个可执行的 COFF 目标文件。当它创建可执行模块时，它实现重定位并解决外部引用问题。链接器接受汇编器所生成的可重定位 COFF 目标文件作为输入，它也接受归档目标库中的成员以及由链接器所创建的输出模块。链接器伪指令允许用户组合目标文件段，把段或符号约束在某地址或存储器范围内，定义全局符号等。

4．归档器(Archiver)

归档器允许用户把多个文件收集到单个归档文件中。用户可以把几个宏源程序收集到宏库内，汇编器会搜索该库并调用源文件中引用的库成员；用户也可以使用归档器把多个目标文件收集到目标库内，链接器会将外部引用的库成员包含到生成的可执行文件中。

5．十六进制转换工具(Hex Conversion Utility)

大多数 EPROM 编程器并不接受 COFF 格式的目标文件。十六进制转换工具把 COFF 格式的目标文件转换成 EPROM 编程器支持的其它格式文件，如 ASCII - hex、TI - Tagged、Intel 或 Motorola - S 格式，转换后的文件就可以被下载到 EPROM 编程器中。

6．绝对列表器(Absolute Lister)

绝对列表器接受链接的目标文件作为输入，生成.abs 文件，这些.abs 文件可被重新汇总从而产生目标文件的绝对地址列表。

7．交叉引用列表器(Cross - Reference Lister)

交叉引用列表器接受目标文件以产生交叉引用列表，此列表显示符号及其定义，并显示在链接的源文件内对它们的引用。

8．运行时支持库创建工具(Library - Build Utility)

利用运行时支持库创建工具，用户可以创建自己的运行时支持库。

9．运行时支持库(Run - Time - Support Library)

运行时支持库包含 ANSI 标准的运行时支持函数、编译器命令函数、浮点算术函数和 I/O 函数等。

10．汇编优化器

汇编优化器仅用于 TI C6000 DSP，它接受用户编写的线性汇编代码作为输入，经汇编优化后生成标准汇编代码。

汇编优化器可自动完成寄存器分配和利用循环优化把线性汇编代码转换成具有高并行度的汇编代码，以充分利用软件流水技术。

用户编写的线性汇编代码不用考虑流水线结构和寄存器分配，这一切由汇编优化器自动完成，因此可以大大降低用户的编程工作量，缩短开发周期。

3.2.2　编译、链接(Build)选项设置

CCS 提供了两种默认的工程级 Build 选项配置：Debug 和 Release。这两种默认工程配置分别为不同的开发过程定义一套 Build 选项，用来控制代码产生(编译、汇编、链接)过程。利用 Debug 配置生成一个供调试用的可执行代码(.out)，称为调试版，并且生成的可执行代码自动放入 Myproject(用户工程路径)\Debug 目录下；利用 Release 配置生成最终 EPROM 固化版本的可执行代码(.out)，称为正式版，并且生成的可执行代码自动放入 Myproject\Release 目录下。

每当创建一个新工程时，Debug 总是作为默认工程配置。可以在 CCS IDE 窗口中选择工程的当前配置，如图 3.6 所示。

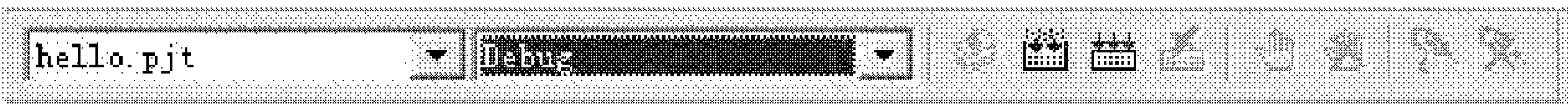


图 3.6　工程配置选择工具条

在 CCS 环境下，用户可以根据需要设置自己的编译器、汇编器、链接器命令开关选项，这些开关选项用来控制编译、汇编、链接过程信息。用户可以设置工程级 Build 选项，工程级选项应用于工程中的所有文件，也可以对工程中的单个源代码文件设置文件级编译选项。CCS 提供了图形对话框方式来设置 Build 选项，这使不熟悉 Build 命令行开关选项的用户也可以在 Build 选项对话框中设置几乎所有的选项。Build 选项对话框也允许用户在文本输入框中直接输入 Build 命令开关选项。

1．设置工程级 Build 选项

(1) 选择(CCS IDE 界面)Project→Build Options，打开工程的 Build Options 对话框，如图 3.7 所示。

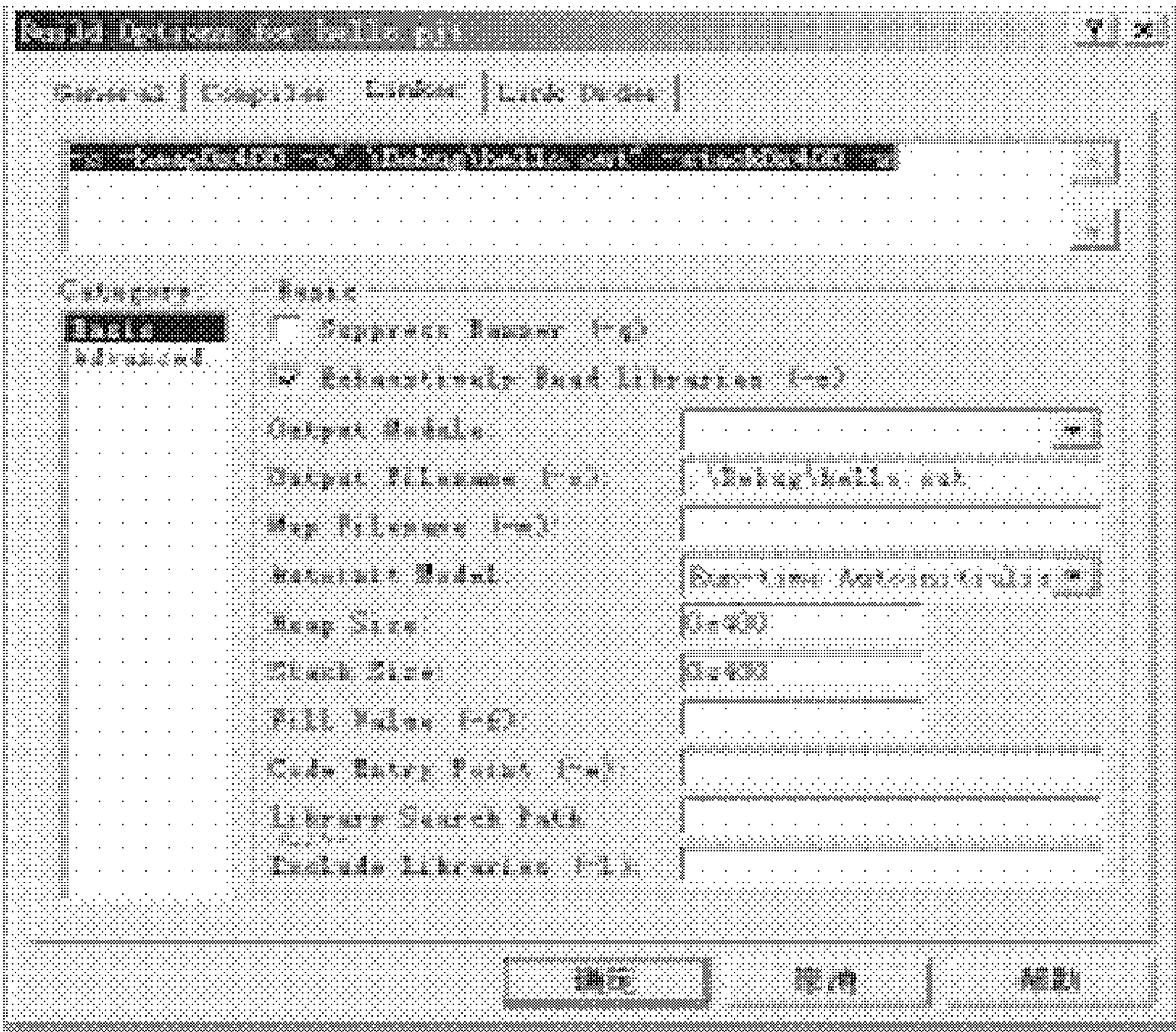


图 3.7　Build Options 对话框

(2) 在 Build Options 对话框中选择需要的编译器、汇编器、链接器开关命令，或直接在文本输入框中输入开关命令。

(3) 点击 OK，接受选项设置并退出 Build Options 对话框。

2．设置文件级编译选项

(1) 右击工程视窗中的源文件名，从弹出的菜单中选择 File Specific Options，打开 File Specific Options 对话框。

(2) 在 File Specific Options 对话框中，设置文件指定的编译选项。

(3) 点击 OK，接受选项设置并退出 File Specific Options 对话框。

为帮助用户熟悉 Build 命令行开关选项，表 3.1、3.2 和 3.3 中分别列出了 C5000 的编译器、汇编器、链接器的命令行开关选项。

表 3.1　编译器的命令行开关选项

选　项	含　义
－pk	允许 K&R 兼容
－pr	使能宽松模式
－ps	使能严格 ANSI 模式(对 C 语言)
－pl	生成原始列表信息文件(*.rl)
－px	生成交叉引用文件(*.crl)
－x0	去除内部链接功能
－x2	使能内部链接用户功能(含有－02 功能)
－d *name*	预定义宏名
－u *name*	未定义的宏名
－as	使汇编器具有－s 选项，从而将符号置入符号列表中
－ga	将每一个变量编译为汇编文件中的全局变量
－aw	使能流水线冲突警告信息显示
－pdw	删除警告信息
－pdv	提供源代码中详细的错误信息
－pdf	生成错误信息文件(*.err)
－o0,…,-o03	设置不同级别的优化级
－oL0,…,oL2	库函数声明选项
－on0,…,on2	编译器生成信息文件详细程度选项
－os	显示 C 的内部列表
－op0,…,op3	告诉编译器模块内变量、函数与原程序外部之间是否有调用修改关系
－ma	假定变量可以有别名
－mn	让由符号级调用－g 禁止的优化使能
－mo	禁止后端(back-end)优化
－ms	优化代码大小而不是速度
－rar1, －rar6	保留 ar1、ar6，使代码生成器和优化器不能用它
－mf	使所有调用和返回都是远程调用和返回(对 c548 或功能更强的 DSP)，缺省时为使用近程调用
－mr	禁止 FPT 指令
－k	保留由编译器输出的汇编文件
－aa	生成绝对汇编列表文件
－al	生成汇编语言列表文件
－ax	生成符号交叉引用列表文件
－ea	允许指定编译器生成的汇编文件扩展名
－fs	允许指定编译器生成的汇编文件目录
－eo	允许指定编译器生成的目标文件扩展名
－fr	允许指定编译器生成的目标文件目录
－ft	存放临时文件的目录

表 3.2 汇编器的命令行开关选项

选 项	含 义
－q	禁止标题和所有信息
－g	使能符号汇编信息
－x	生成列表文件
－pw	生成流水线冲突信息
－mf	指定使用远程汇编调用
－c	使汇编程序中的大小写等效
－s	将所有已定义的符号放在目标文件的符号表中
－d *name*	预定义宏名
－u *name*	未定义宏名
－hc	告诉汇编器，汇编模块拷贝的指定文件
－hi	告诉汇编器，汇编模块包括的指定文件
－f	禁止将 asm 扩展名加入无扩展名的源文件名后

表 3.3 链接器的命令行开关选项

选 项	含 义
－q	不显示标题和所有信息
－a	产生绝对的可执行模块输出
－r	产生可重定位的输出模块输出
－o *filename*	指定输出的可执行文件名 *filename.out*
－m *filename*	产生输入和输出段的映像或列表，并把列表放在 *filename.map* 文件中
－e *global_symbol*	产生 *global_symbol*(全局符号)，它规定输出模块的主入口点
－c	使用编译器 ROM 自动初始化模块定义链接约定
－cr	使用编译器 RAM 自动初始化模块定义链接约定
－j	禁止在汇编器内设置的条件链接
－k	强制链接器忽略段指令对齐的约定
－heap *size*	把堆(用于 C/C++语言中动态存储器分配)的大小设置为 *size* 个字并定义规定堆大小的全局符号
－stack *size*	把 C/C++系统堆栈大小设置为 *size* 个字并定义规定堆栈大小的全局符号
－f *fill_value*	为输出段中的空余存储器设置缺省填充值(*fill_value*)
－b	禁止符号调试信息的合并
－s	从输出模块中删除符号表信息和行号入口
－h	使所有全局符号为静态
－u *symbol*	把未定义的外部符号放入输出模块的符号表中
－x	使链接器重新读库，解释反向引用
－i *dir*	改变库搜索算法以便在搜索缺省目录之前搜索 *dir*
－l *filename*	命名作为链接器输入的归档文件，*filename* 是归档库名

3.2.3　代码产生过程演示例子

本节利用 CCS 安装目录中提供的一个例子，来演示创建新工程、编译链接生成可执行代码、加载运行的整个详细过程。

如果 CCS 安装在 c:\ti\目录下，创建一个新的用户目录 c:\ti\myproject\hello1\，把 c:\ti\tutorial\dsk6711\hello1\ 目录中的所有文件(除去工程文件.pjt)拷贝到 c:\ti\myproject\hello1\目录下。(本章以 C6711 DSK 目标板为例，因此 CCS 提供的 hello1 例子在 c:\ti\tutorial\dsk6711\hello1\目录下。本章的演示过程可应用于 CCS 支持的所有调试器。)

1．创建一个新工程

(1) 选择(CCS IDE 界面)Project→New，打开 Project Creation 对话框，如图 3.8 所示。



图 3.8　Project Creation 对话框

(2) 在 Project Creation 对话框的 Project Name 文本输入框中输入新建的工程名 myhello。

(3) 在 Location 项中指定工程目录：c:\ti\myproject\hello1\。

(4) 在 Project Type 项中选择 Executable(.out)。

Project Type 有两种选项：

- Executable(.out)：把工程编译链接生成一个可执行文件。
- Library(.lib)：生成一个目标库。

(5) 在 Target 项中选择用户的目标 DSP 类型。

(6) 点击完成按钮，退出 Project Creation 对话框。CCS IDE 会创建一个工程文件 myhello.pjt 并保存在 c:\ti\myproject\hello1\目录下，并且 myhello.pjt 会自动作为 CCS IDE 的当前工程显示在 CCS IDE 的工程视窗内。myhello.pjt 具有默认的 Debug 编译链接配置。

2．向工程中添加文件并查看文件

(1) 选择 Project→Add Files to Project(或右击工程视窗中的 myhello.pjt，从弹出的菜单中选择 Add Files)，在弹出的 Add Files to Project 窗口中选择适当的文件类型，并将相应的文件添加到工程中，这些文件包括(c:\ti\myproject\hello1\目录下)：

- hello.c 文件(文件类型选为*.c)，该文件为 C 语言源代码主程序。
- vectors.asm 文件(文件类型选为*.a*)，此汇编文件定义中断服务程序，其中包含控制

转向 C 程序入口点_c_int00 的 RESET(复位)中断服务指令包。

- hello.cmd 文件(文件类型选为*.cmd)，此文件是链接命令文件，用来分配存储器段。
- rts6700.lib 文件(文件类型选为*.lib*)，此文件在 c:\ti\c6000\cgtools\lib\目录下，提供目标板的运行时支持。

(2) 选择 View→Project，打开工程视窗(如果先前没有打开)，点击工程视窗中 Project 左侧的“＋”号，将工程中所包含的工程文件(*.pjt)、链接命令文件(*.cmd)、库函数(*.lib)、源代码(*.c,*.asm)以树的形式显示出来，如图 3.9 所示。

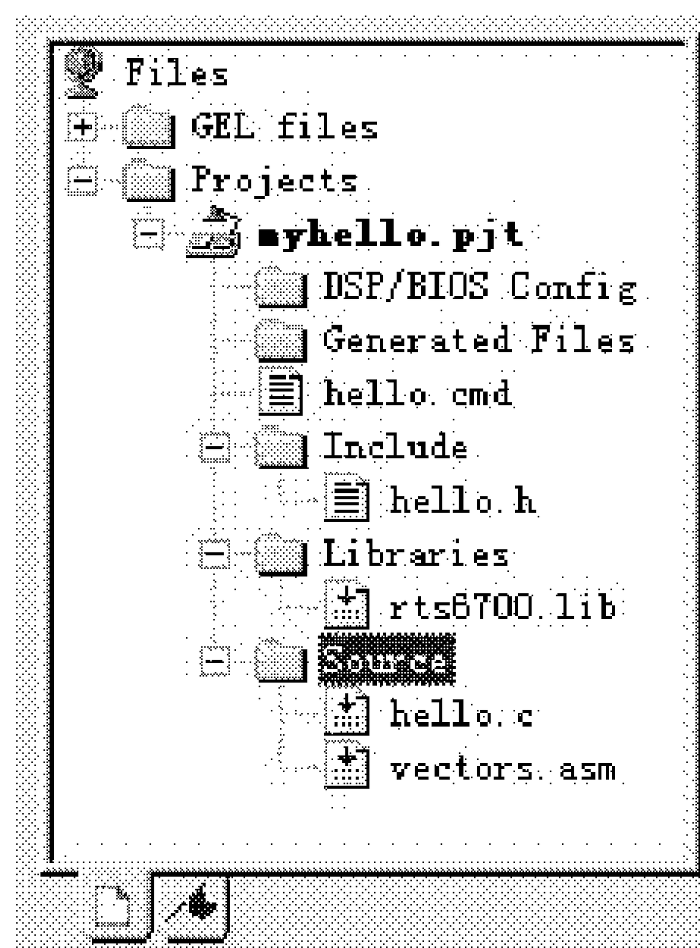


图 3.9 CCS 工程视窗

CCS IDE 会自动按文件类型把加入的文件分配到不同的文件夹中。

注意，头文件(include file)不需要用户手动加入工程中，CCS IDE 在对工程进行编译链接(Build)过程中，将源程序中出现的 include 文件进行搜索并自动添加到工程视窗的 Include 文件夹下。

如果需要删除工程中的某个文件，右击工程视窗中的该文件，从弹出的菜单中选择 Remove from Project 即可。

(3) 双击工程视窗中的 hello.c 文件，即可在右侧的文档窗口中打开该文件，这是 CCS IDE 提供的源代码编辑工具，在该窗口内实现对源代码的编辑操作。如果需要编写一个新的源代码文件，选择 File→New→Source File，在打开的新编辑窗口中进行源代码编写，之后按适当的类型保存该文件。hello.c 文件的源代码如下：

```
#include <stdio.h>
#include "hello.h"
#define BUFSIZE 30
struct PARMS str =
{
   2934,9432,213,9432,&str
};
/* ======== main ======== */
void main()
{
  #ifdef FILEIO
      int          i;
      char         scanStr[BUFSIZE];
      char         fileStr[BUFSIZE];
      size_t       readSize;
      FILE         *fptr;
  #endif

  /* write a string to stdout */
    puts("hello world!\n");
```

```
    #ifdef FILEIO
    /* clear char arrays */
        for (i = 0; i < BUFSIZE; i++) {
            scanStr[i] = 0              /* deliberate syntax error */
            fileStr[i] = 0;
        }

    /* read a string from stdin */
    scanf("%s", scanStr);

    /* open a file on the host and write char array */
    fptr = fopen("file.txt", "w");
    fprintf(fptr, "%s", scanStr);
    fclose(fptr);

    /* open a file on the host and read char array */
    fptr = fopen("file.txt", "r");
    fseek(fptr, 0L, SEEK_SET);
    readSize = fread(fileStr, sizeof(char), BUFSIZE, fptr);
    printf("Read a %d byte char array: %s \n", readSize, fileStr);
    fclose(fptr);
  #endif
}
```

3．编译链接工程生成可执行代码并加载运行

1) 编译链接(Build)工程

选择 Project→Rebuild All 或点击工具栏中的 Rebuild All 图标，CCS IDE 将对工程中的所有文件进行重新编译、重新汇编，并重新链接全部目标文件。生成的过程信息将在自动弹出的信息显示窗中显示出来。生成的可执行文件 myhello.out 在 c:\ti\myproject\hello1\Debug\ 目录下。

2) 加载可执行代码

选择 File→Load Program，在弹出的窗口中选择 c:\ti\myproject\hello1\Debug\myhello.out 文件，点击 Open 即可。CCS 将该可执行代码加载到目标 DSP 中，并同时打开反汇编窗口(Disassembly)，显示出目标程序所对应的反汇编指令(在反汇编窗口中，若高亮选中某条汇编指令，然后按下 F1 键，CCS 将提供该指令的在线帮助，这对初学者来说非常方便)。经编译、汇编、链接并把可执行代码加载后，就可以将该工程运行了。

3) 运行加载在 DSP 中的可执行代码

选择 Debug→Run 或点击工具栏中的 Run 图标，运行目标 DSP 中的该程序。运行结果“hello world!”显示在标准输出 stdout 窗口中，如图 3.10 所示。

选择 Debug→Halt 或点击工具栏中的 Halt 图标，停止目标 DSP 中程序的运行。



图 3.10　Myhello 运行结果显示

4) 改变工程的编译链接命令行选项并修改语法错误

如前所述，利用 CCS 创建一个新工程时，CCS 提供了两种默认的工程级编译链接配置：Debug 和 Release。CCS 允许用户设置自己的编译链接命令行选项。

由于 FILEIO 符号没有事先定义，因此上例 hello.c 文件中预处理命令 # ifdef FILEIO 和 # endif 之间的程序段没有被编译链接，实际上 DSP 只执行了 hello.c 主函数的一句程序，即 puts("hello world!\n")。

按下列步骤完成定义 FILEIO 和语法错误纠正：

(1) 选择 Project→Build Options，打开 Build Options 对话框。

(2) 点击 Build Options 对话框的 Compiler 栏，打开 Compiler 的选项面板。

(3) 在 Compiler 选项面板的 Category 中选择 Preprocessor，可以看到面板左侧 Define Symbols(－d)：的文本输入框中有预定义符号_DEBUG，再在此文本输入框中输入 FILEIO 并与_DEBUG 以分号相隔，此时可注意到，在命令行显示窗口内自动添加了 －d "FILEIO" 命令项。

(4) 点击 OK，保存新的 Build 选项设置，并重新对整个工程进行编译链接(Rebuild All)。

(5) 编译过程会显示出错信息：在第 52 行少了一个分号。

(6) 双击出错信息，CCS 会自动打开 hello.c 文件(如果先前没有打开)，并且鼠标停在出错行。

(7) 改正语法错误：在第 52 行语句 scanStr[i]=0 后添加一分号“；”，将修改过的文件存盘后，选择 Project→Build，CCS 只对上一次编译链接后被修改的文件进行重新编译链接，生成可执行代码。

(8) 重新加载目标 DSP 并运行新生成的可执行代码。

至此，我们已经可以在 CCS 中对应用工程进行创建、编译、汇编、链接、加载运行以及设置编译链接的命令行选项，下一节我们将详细介绍如何在 CCS 中对应用程序进行调试。

3.3 代码调试工具

CCS IDE 集成了多种代码调试工具用来帮助开发者快速发现程序中的错误、瓶颈等问题，实际上代码调试包括两种：一种是程序的逻辑功能调试，即程序是否能完成设计要求的功能；另一种是程序的实时性调试，即调试与时间有关的问题，例如，程序段的执行时间是否满足要求，程序中各线程的调用是否存在瓶颈现象等。本节主要介绍 CCS IDE 提供的逻辑功能调试工具，在 3.4 节中将详细介绍 CCS IDE 提供的实时分析工具，这是 CCS IDE 不同于传统调试工具的特色之处。

3.3.1 CCS 提供的调试工具

1. 断点和程序运行控制

CCS 提供了完善的程序运行控制功能，主要有以下几点：

1) 断点(Breakpoint)

当程序运行到断点时，会停止程序的执行，此时开发人员可以查看程序的运行状态、检查和修改变量、观察堆栈调用情况等。断点可以在编辑窗中源文件的某一行上设置，也可以在反汇编窗中的某一条指令上设置。CCS IDE 提供了三种断点：软件断点(Software Breakpoint)、硬件断点(Hardware Breakpoint)和条件断点(即当某一条件为真时，才产生断点)。

断点可以在任何编辑窗或反汇编窗中设置，而且断点的数目也不受限制。

设置断点最简单的方法就是在编辑窗或反汇编窗中用鼠标双击需要设置断点的行，或选定指定行后点击工具栏中的 Toggle breakpoint 图标，或点击鼠标右键从弹出的菜单中选择 Toggle breakpoint。CCS 会在已设置断点的行前用一个红圆点作为标识。

设置断点的另一种方法稍微复杂一些，但可以设置多种类型的断点(软件断点、硬件断点和条件断点)，这种方法可按如下步骤实现：

(1) 选择 Debug → Breakpoints，打开 Break/Probe Points 对话框(设置断点和探点)，如图 3.11 所示。

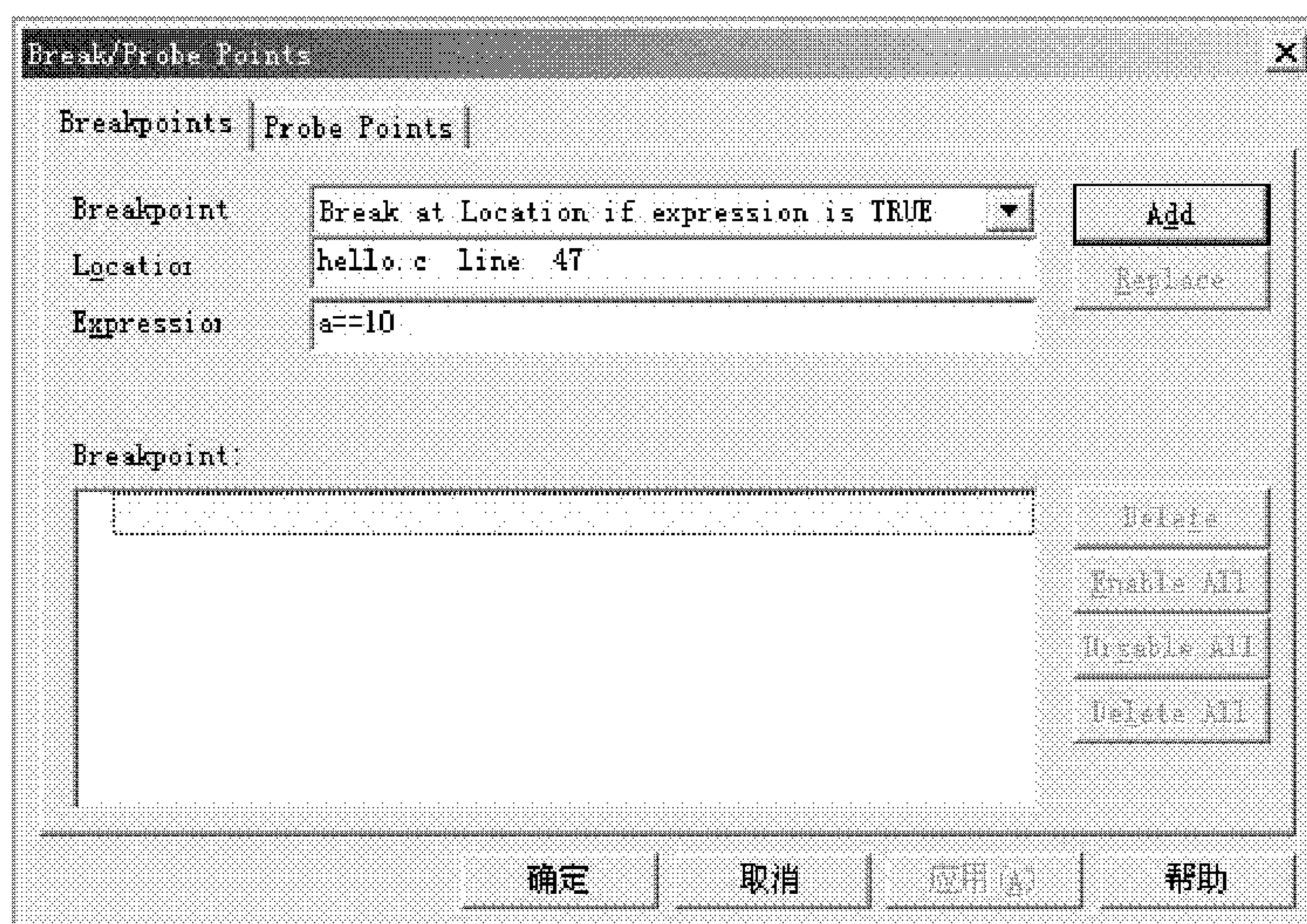


图 3.11　Break/Probe Points 对话框

(2) 打开 Break/Probe Points 对话框的 Breakpoints 面板栏。

(3) Breakpoints 项中有多种选择：Break at Location(设置软件断点)、Break at Location if expression is TRUE(设置条件断点，即当指定的表达式为真时，程序运行到此行才会停止，否则程序会继续运行)、H/W Break(设置硬件中断，当此行的指令被 DSP 访问时产生断点，而且可以设置计数器(Count 项)，即当指定的访问次数到达时才会产生中断)。

(4) 在 Location 项中指定断点的位置，有两种方法：

- 指定绝对地址，即在 Location 项中输入 C 表达式、C 函数名或符号名。
- 指定源文件的行，即在 Location 项中输入：文件名 line 行数

例　hello.c　line　47

上例表示在 hello.c 文件的第 47 行设置断点。

(5) 在 Expression 项中输入 C/C++表达式，当此表达式为真时条件断点才会停止程序执行。只有当 Breakpoint 项中选择 Break at Location if expression is TRUE 时，此项才被激活。

例如，在 Expression 项中输入：

例　a= =10

上例中，a 为 C 变量名。当程序运行到 Location 中指定位置时，如果 a 等于 10，则 CCS 会停止程序执行，如果 a 不等于 10，则程序会继续执行。

(6) 在 Count 项中指定某一数字。当 DSP 对某一指令的执行次数超过 count 中指定的次数时，CCS 才会在此指令处停止程序的执行。只有当 Breakpoint 项中选择 H/W Break 时，此项才被激活。

(7) 点击 Add，添加一个前面创建的新断点。

(8) 点击 OK，退出 Break/Probe Points 对话框。

接着就可以利用其它的调试工具或显示工具来调试程序了。

删除断点：双击已设置断点的行，或点击工具栏中的 Remove all breakpoints 图标来删除所有断点，或在 Break/Probe Points 对话框中删除某一个或所有断点。

2) 动态运行程序(Animate)

启动 Animate 后，开始运行程序，到达某一断点处后更新所有窗口，然后继续运行，直到到达另一断点处，如此反复下去，直到用户停止程序的执行。

Animate 到达某一断点处后，暂停的时间受 Animate Speed 控制，Animate Speed 可由用户指定：

(1) 选择 Option(CCS IDE 界面菜单)→Customize，打开 Customize 对话框。

(2) 打开 Customize 对话框的 Animate Speed 面板栏，在 Animate Speed 项中输入一个 0～9 的数字(单位：秒)。

开始动态执行程序：选择 Debug→Animate 或点击调试工具栏中的 Animate 图标。

停止 Animate 的执行：选择 Debug→Halt 或点击调试工具栏中的 Halt 图标。

3) 其它控制程序运行工具

• 跳过函数体：选择 Debug→Step Over 或点击调试工具栏中的 Step Over 图标。

• 从函数体跳出：选择 Debug→Step Out 或点击调试工具栏中的 Step Out 图标。

• 单步执行(执行下一行指令后停止，可以为 C/C++指令或汇编指令)：选择 Debug→Step Into 或点击调试工具栏中的 Single Step 图标。

• 运行至光标所在行：选择 Debug→Run to Cursor 或点击调试工具栏中的 Run to Cursor 图标。

• 多次执行(Step Into、Step Over、Step Out)操作：选择 Debug→Multiple Operation，打开 Multiple Operation 对话框，如图 3.12 所示。在 Multiple Operation 对话框中选择步执行命令并指定执行次数，点击 OK 退出对话框后，程序即按要求开始执行。

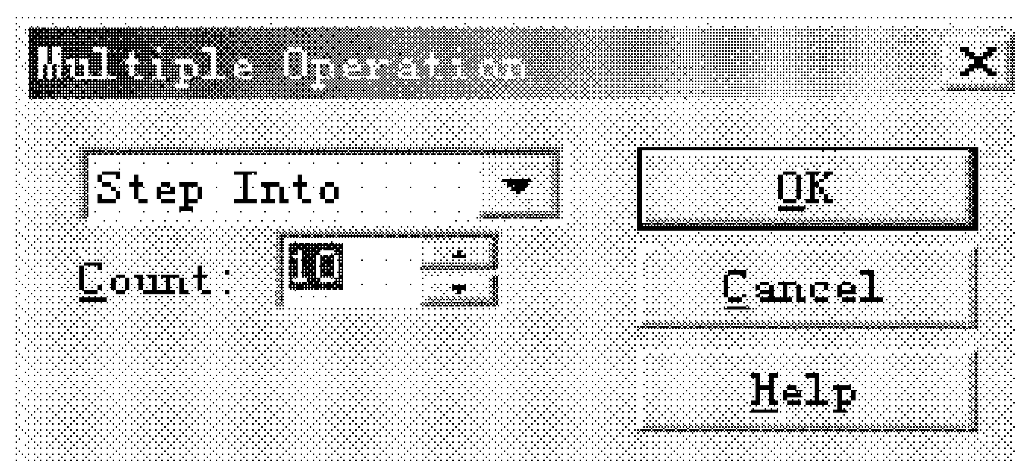


图 3.12　Multiple Operation 对话框

• 连续运行(忽略掉所有断点和探点)：选择 Debug→Run Free。

4) 复位程序工具

• 复位 CPU(复位所有寄存器到上电时的初始状态，停止程序执行)：选择 Debug→Reset CPU。

• 恢复程序计数器 PC 到当前程序的入口点：选择 Debug→Restart。

• 从当前程序计数器 PC 的位置处运行到主函数 main( )的入口点，然后停止程序执行：选择 Debug→Go Main。

2．探点(Probepoints)

探点在进行算法调试时是非常有用的，利用探点可以完成如下功能：

• 实现从主机文件向目标 DSP 的存储器输入数据。

• 实现从目标 DSP 的存储器向主机文件输出数据。

• 程序执行到探点处会更新指定窗口。

探点类似于断点，它们都可以中止程序的执行，但又有所不同：

• 探点只是暂时停止当前程序的执行，当完成探点的任务后，继续执行接下来的程序。

• 断点会一直停止 DSP 的执行，直到用户使用运行或单步执行命令后才可以使 DSP 重新运行，并且断点会更新所有打开的窗口。

• 探点可以实现数据从主机文件的自动输入和输出，而断点则不能。

按下列步骤可完成探点设置和数据文件的输入、输出：

(1) 选择 File→Load Program，把编译链接后生成的可执行文件加载到目标 DSP 中。

(2) 双击工程视窗中的源文件 filename.c，在编辑窗中打开此源文件，并把光标移动到需要设置探点的行上。

(3) 点击工具栏中的 Toggle Probe Point 图标，在此行设置一个探点，此行的前面会出现一个绿色棱形，标识此行已设置了一个探点。

(4) 选择 File→File I/O，打开 File I/O 对话框，如图 3.13 所示。

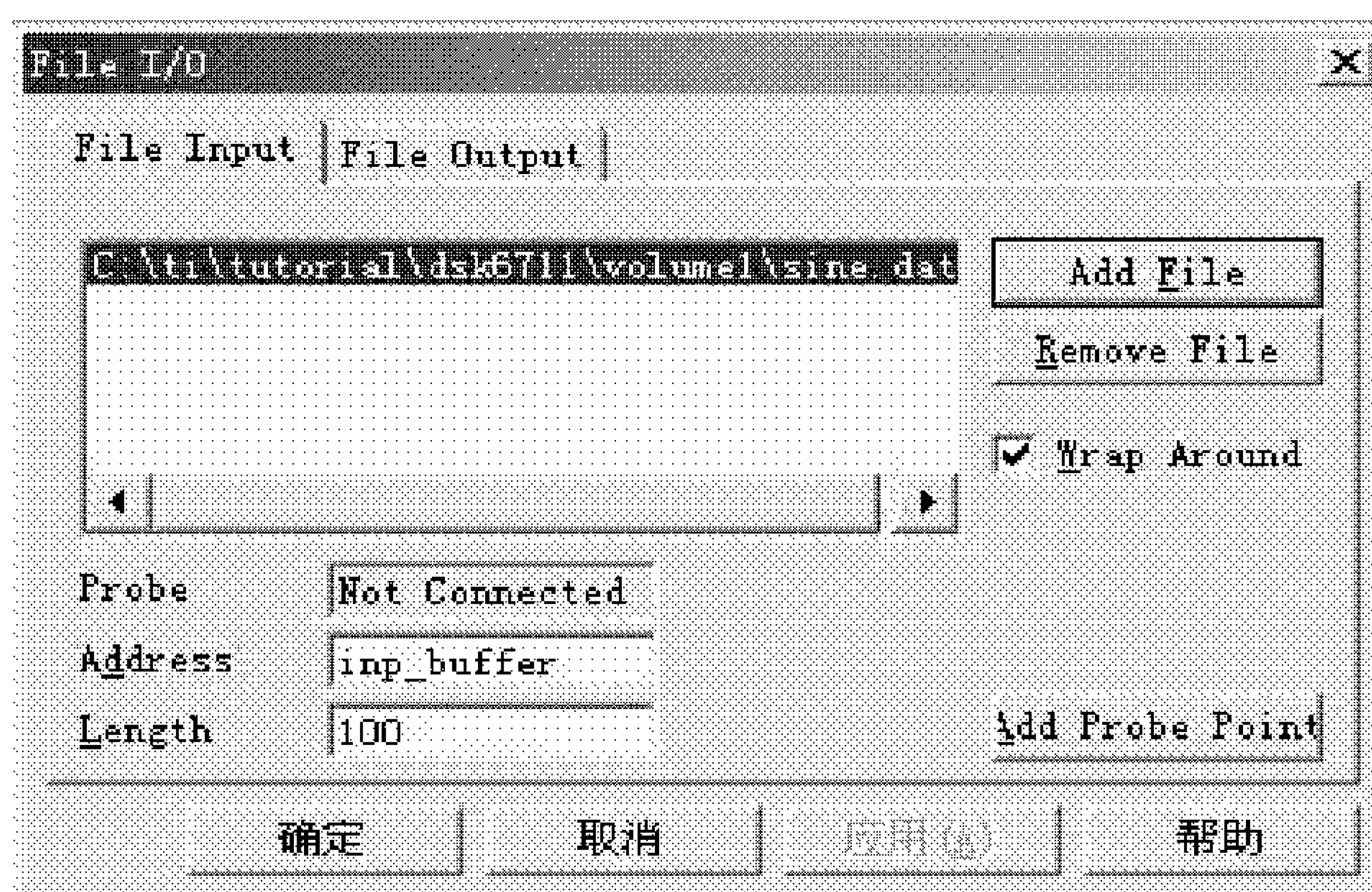


图 3.13　File I/O 对话框

(5) 在 File Input(文件输入)面板中点击 Add File，选择输入数据的文件名，并打开此文件，会出现一个文件数据流控制工具条，如图 3.14 所示。

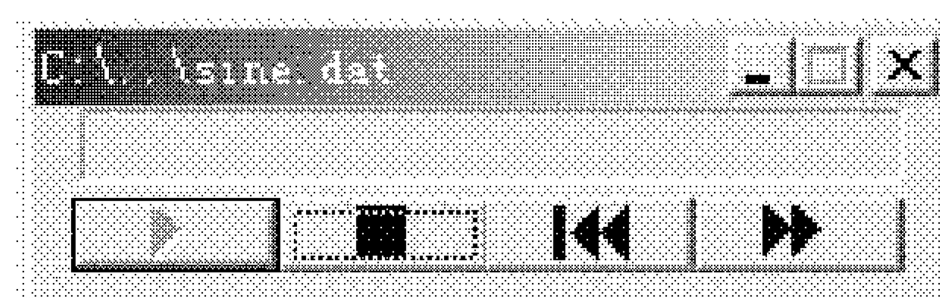


图 3.14　文件数据流控制工具条

利用此文件数据流控制工具条，可以开始、停止、倒回和快速前进从文件到目标 DSP 的数据输入。

(6) 在 File I/O 对话框中，设置 Address、Length 和 Wrap Around 项。

• Address：指定目标 DSP 的存储器中放置此数据的首地址。

• Length：指定每当程序运行到此探点处时，从数据文件读入的数据长度。

• Wrap Around：如果选择此项，则当读完此文件中的所有数据后，CCS 重新从文件的开头继续读。

(7) 点击 Add Probe Point，打开 Break/Probe Points 对话框的 Probe Points 面板，如图 3.15 所示。

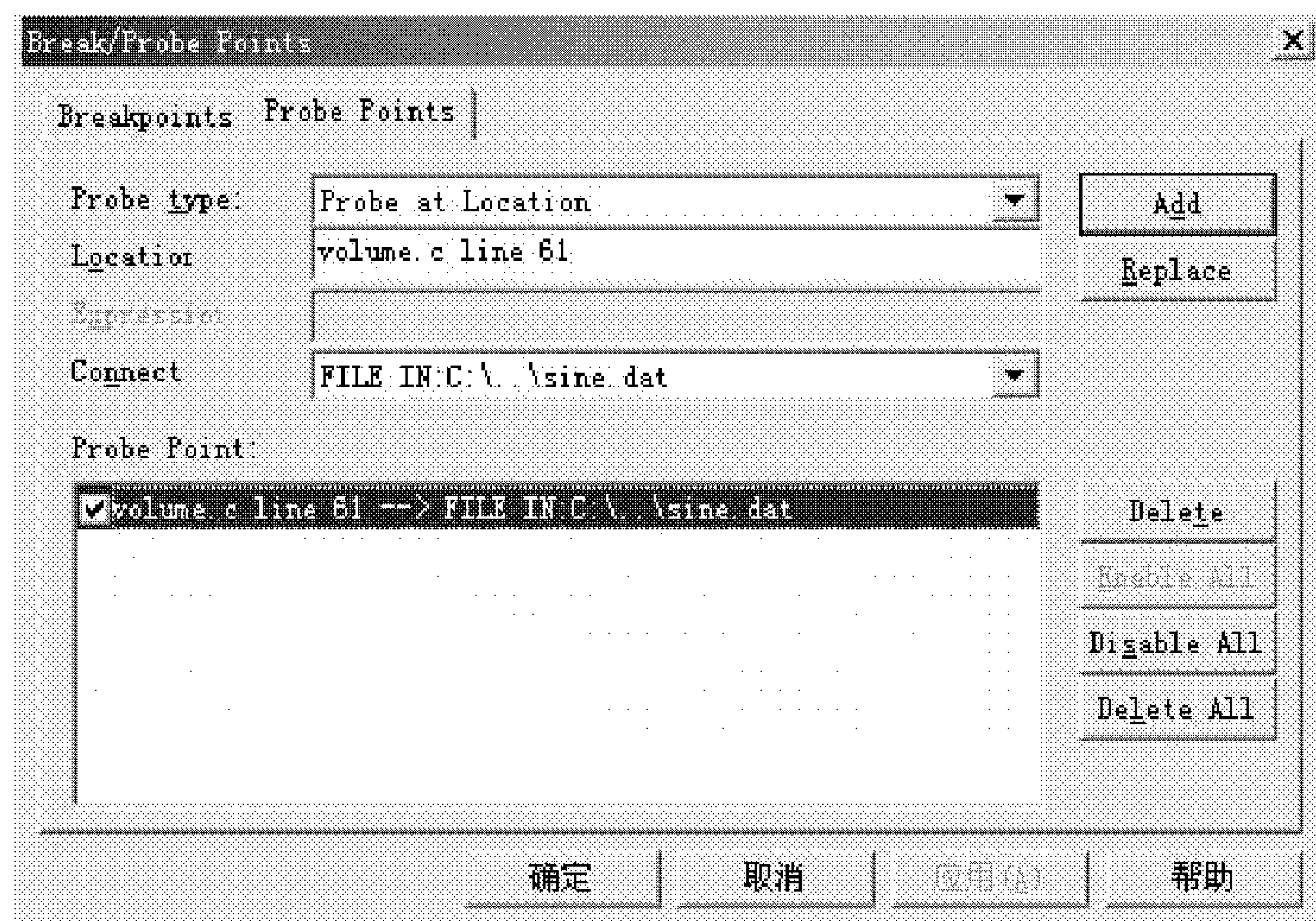


图 3.15　Break/Probe Points 对话框的 Probe Points 面板

(8) 在 Probe Points 面板的 Probe Point 列表中高亮选择某一需要的探点。

(9) 在 Connect To 项中，从列表中选择数据文件。

(10) 点击 Replace 按钮，从 Probe Point 列表中就可以看到探点与数据文件连接起来了。

(11) 点击 OK，退出 Break/probe Points 对话框。

(12) 点击 OK，退出 File I/O 对话框。

(13) 程序运行后，当到达探点处时会自动从主机文件输入数据。

(14) 如果要删除所有探点，则点击工具栏中的 Remove all Probe Points 图标；如果只要删除某一探点，则只要再点一下 Toggle Probe Point 图标就可以了。

在探点处设置输出文件(数据从目标 DSP 输出到主机的文件中)与上述设置输入文件的步骤相同。

探点也有多种类型，可从图 3.15 的 Probe type 项中进行选择：

• Probe at Location：软件探点，在源文件的行或绝对地址处设置一探点。

• Probe at Location if expression is TRUE：条件探点，当程序运行到此行时，如果表达式为真，则执行探点操作；如果表达式为假，则程序会继续执行下一条指令。

• H/W Probe at Location：设置硬件探点，当设置探点的指令行被 DSP 执行的次数达到 Count 中指定的次数时，程序就会执行探点任务。

这些探点类型的设置方法与前面断点的设置方法类似，这里不再详述。

利用探点不但可以与主机文件进行数据的输入/输出，还可以利用探点来更新某一窗口，即程序到达探点处后，更新这一窗口，然后继续运行。按下列步骤可完成探点与某一窗口的连接：

(1) 在 CCS IDE 中打开需要观察的窗口，例如，Watch 窗、存储器或寄存器观察窗口。

(2) 在源文件或反汇编程序的指定行设置一探点。

(3) 选择 Debug→Probe Points，打开 Break/Probe Points 对话框。

(4) 打开 Probe Points 面板栏。

(5) 在 Probe Points 探点列表中，高亮选中前面所设置的探点(此时可以看到，此探点指示为没有连接)。

(6) 设置好 Probe type、Location、Expression 和 Count 选项(根据不同的探点类型进行设置，其方法类似于断点)。

(7) 从 Connect 列表中选择某一打开的观察窗口。

(8) 点击 Replace，此时在探点列表中可以看到所选的探点与指定显示窗口连接起来了。

(9) 点击 OK，退出 Break/Probe Points 对话框。

至此，我们就完成了探点与指定窗口的连接。当程序运行到此探点处时，更新这一窗口，然后继续运行，而到达断点处时会更新所有窗口，但需要手动才能继续运行。

3．显示和观察窗口

CCS IDE 提供了多种显示窗口，可以用来显示和修改源代码、反汇编代码、存储器和寄存器中的值、C 程序中的变量值等，而且还可以观察堆栈调用情况、对数据进行图形显示等。

以下详细介绍 CCS IDE 提供的显示功能。

1) 源代码编辑窗口

源代码编辑窗口用来查看、修改和编辑任意文本文件。在调试过程中，源代码编辑窗口用来显示当前加载程序的源代码以及监视调试进程的信息(CCS IDE 的调试工具与源代码编辑窗口紧密连接在一起)。

双击工程视窗中的源文件，系统会自动在源代码编辑窗中打开此文件。源文件编辑窗是 CCS IDE 的一个基本窗口，在调试过程中会经常用到，这里不再详述。

2) 反汇编窗口

把可执行代码加载到目标板或 Simulator 中时，CCS IDE 会自动打开反汇编窗口。反汇编窗中显示反汇编指令及符号信息。对每一条反汇编指令，反汇编窗同时显示此指令所在的存储器地址及其对应的机器代码。用一个黄色箭头指示当前程序计数器 PC 的位置。

可以同时打开多个反汇编窗口，每打开一个反汇编窗口，会在其标题栏中显示：Disassembly<n>，用来指示打开的第 n 个反汇编窗口。第 1 个打开的反汇编窗口(即 Disassembly<1>)总是随着当前程序计数器 PC 位置的变化而变化，即它总是显示当前程序计数器的位置，而其它打开的反汇编窗口用来固定显示某一存储器地址处的代码段。

打开反汇编窗口。选择 View→Disassembly，或点击调试工具栏中的 View disassembly 图标。

改变反汇编窗口中显示的存储器段。在反汇编窗口中点击鼠标右键，从弹出的菜单中选择 Start Address，在 View Address 对话框中输入绝对地址或 C/C++符号的表达式。

反汇编窗也允许用户设置其显示模式。选择 Option→Disassembly Style，或点击反汇编窗口，从弹出的菜单中选择 Properties→Disassembly Options，在 Disassembly Style Options 对话框中设置反汇编窗的显示模式。

3) 变量或 C/C++表达式显示窗口(Watch Window)

利用 Watch Window 窗可以显示程序中的局部和全局变量以及 C/C++表达式。

选择 View→Watch Window 或点击工具栏中的 Watch Window 图标，打开 Watch Window 窗口，如图 3.16 所示。

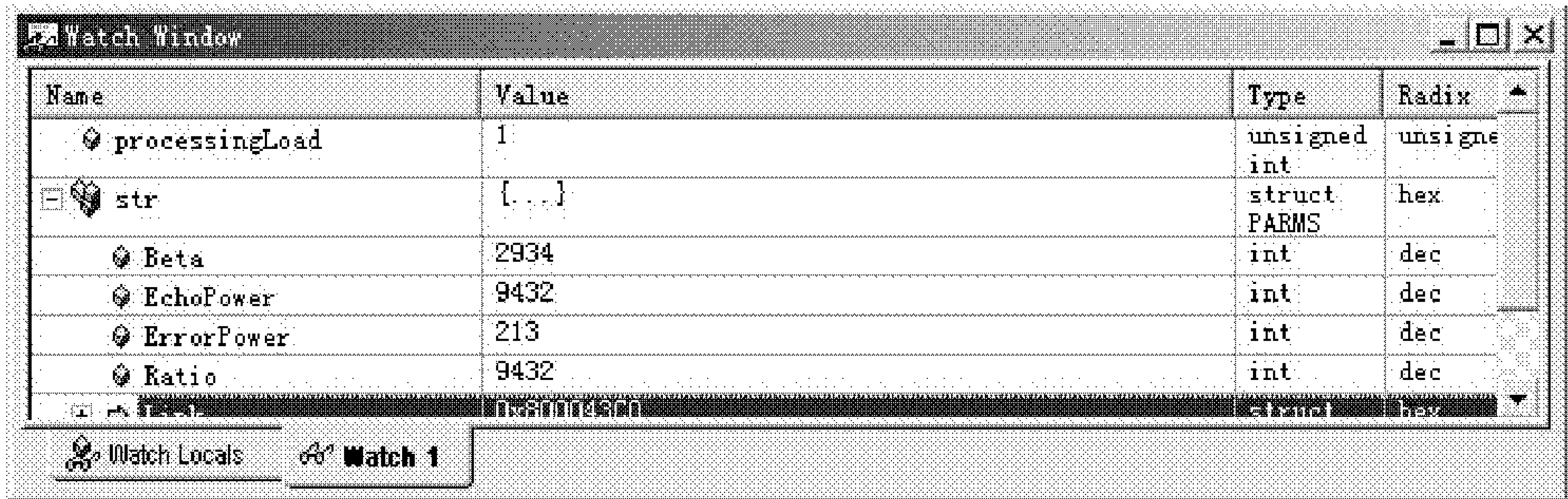


图 3.16 Watch Window 显示窗口

图 3.16 的 Watch Window 窗口包含两部分：

• Watch Locals：CCS IDE 在此子窗口内会自动显示当前执行指令所在函数体内的所有局部变量的信息，包括变量名、变量值以及数据类型等。

• Watch：在此子窗口内用户必须输入要查看的全局变量、局部变量名，CCS IDE 会显示出这些变量所对应的变量值、数据类型等。

在 Watch 栏中输入变量名(全局变量或局部变量)的方法很简单：在 Name 列的文本输入框中输入变量名或表达式即可，CCS IDE 会显示这些变量或表达式的值及其数据类型。

另一种显示变量或表达式的方法是：选择 View→Quick Watch 或点击 Quick Watch 图标，打开 Quick Watch 对话框，在其文本输入框中输入变量名或表达式，点击 Add To Watch，就会把此变量或表达式添加到 Watch Window 窗中；也可直接在编辑窗中高亮选择此变量，然后右击鼠标，从弹出的菜单中选择 Add to Watch Window。

在 Watch Window 窗中可以修改变量值：点击变量的 Value 项，修改其值，然后点击其它的空白处，就完成了变量值的修改，我们可以看到，修改后的值暂时以红色标识。

关闭 Watch Window 窗：右击此窗口，从弹出的菜单中选择 Close 就可以了。

4) 存储器显示窗口(Memory Window)

利用存储器显示窗口可以显示存储器中的内容，并可以对存储器内容进行修改。按下列步骤可以实现对一段存储器内容进行显示和修改的功能：

(1) 选择 View→Memory 或点击调试工具栏中的 View Memory 图标，打开 Memory Window Option 对话框，如图 3.17 所示。

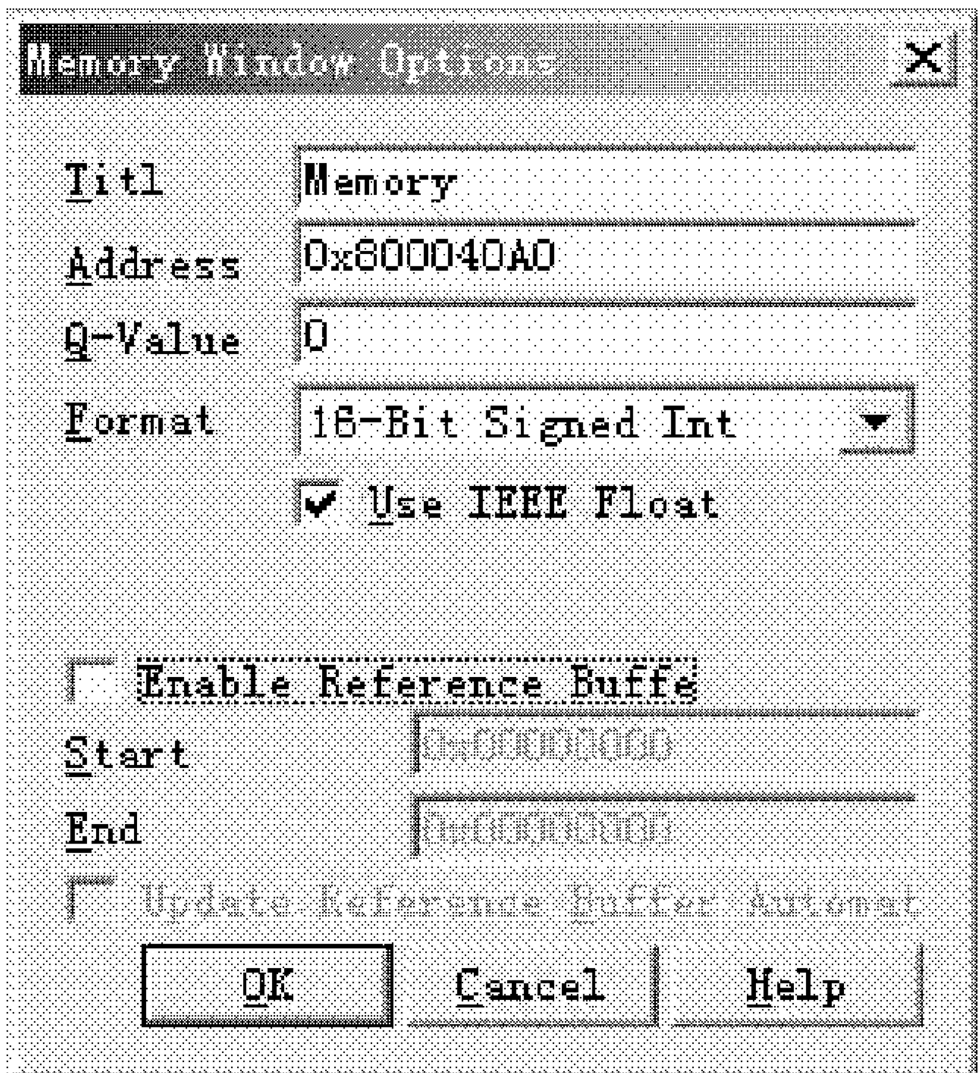


图 3.17 Memory Window Option 对话框

(2) 在 Memory Window Options 对话框中设置存储器显示窗口的各种选项。Memory Window

Options 对话框中有如下选项(不同的目标平台，其选项有所不同)：

- Title：在此项中输入存储器显示窗口的标题名，此名会显示在存储器窗口的标题栏处，用以区分多个存储器显示窗口。
- Address：在此项中输入所要显示的存储器内容的开始地址。
- Q - Value：在此项中指定二进制小数点的位置(从最低位开始，0～31 之间的某一数字)，即利用 Q 格式定点类型来显示数据。
- Format：从下拉菜单中选择存储器内容的显示格式。
- Use IEEE Float：如果选择此项，则利用 IEEE 浮点格式来显示存储器内容。
- Enable Reference Buffer：如果选择此项，则由下面的 Start Address 和 End Address 项指定的一段存储器内容会保存到主机的缓冲区中。每当用户停止 DSP 的执行，或碰到一个断点，或刷新存储器显示窗口时，CCS IDE 会把当前存储器中的内容与保存在主机缓冲区中的内容进行比较，改变值用红色标识出来。
- Start Address：指定要保存的存储器段的首地址。此项只有当 Enable Reference Buffer 被选择时，才被激活。
- End Address：指定要保存的存储器段的末地址。此项只有当 Enable Reference Buffer 被选择时，才被激活。
- Update Reference Buffer Automatically：如果选择此项，每当用户停止 DSP 的执行，或碰到一个断点，或刷新存储器显示窗口时，当前存储器段的内容会自动替换主机缓冲区中的内容；如果不选择此项，主机缓冲区中的内容不会被改变。此项只有当 Enable Reference Buffer 被选择时，才会被激活。

(3) 点击 OK，退出 Memory Window Options 对话框，同时打开存储器显示窗口。

(4) 修改存储器内容，在存储器显示窗口中双击需要修改的地址或内容，会弹出 Edit Memory 对话框，此对话框的 Address 项为存储器地址，Data 为地址处的内容，修改 Data 中的值，然后点击 Done 退出 Edit Memory 对话框，这样就修改了此地址处的内容。开发者不但可以修改单个地址处的内容，还可以用某数值填充一段存储空间，甚至还可以把某段存储空间的内容复制到另一段存储空间中，具体做法是：

选择 Edit→Memory→Fill，打开 Setup Filling Memory 对话框，在此对话框中指定填充的数值及用此数值填充的存储段范围。

选择 Edit→Memory→Copy，打开 Setup for Copying 对话框，在此对话框中指定复制的源存储段和目的存储段。

如果要关闭此显示窗口，右击此窗口，从弹出的菜单中选择 Close 就可以了。

5) 寄存器显示窗口

利用寄存器显示窗口来显示 CPU 核寄存器或外设寄存器的内容，而且可以修改这些寄存器的内容。

显示寄存器内容：选择 View→CPU Registers→Core Registers(核寄存器)、Peripheral Register(外围寄存器)、DMA Registers(DMA 寄存器)、Serial Port Register(串口寄存器)等，也可以直接点击调试工具栏中的 Register Windows 图标来打开 CPU 核寄存器显示窗口。

修改寄存器内容：在寄存器显示窗口中双击某一寄存器，或点击鼠标右键，从弹出的菜单中选择 Edit Register，或选择 Edit→Register，在打开的 Edit Registers 对话框中修改寄

存器内容。

6) 堆栈调用窗口(Call Stack Window)

利用此窗口观察当前函数的调用情况(即此窗中列出了所有当前正在调用的函数名)，用于确定当前程序停止处所在的位置。

选择 View→Call Stack 或点击调试工具栏中的 View Stack 图标，打开堆栈调用窗口(Call Stack)。

双击堆栈调用窗口中的某一函数名，CCS IDE 会自动打开此函数的源代码并且光标会停在当前执行的位置(行)处。

7) 符号浏览窗口(Symbol Browsers)

利用符号浏览窗口来显示已加载可执行代码中的所有源文件、函数、全局变量、定义的类型和标号。

打开符号浏览窗口：选择 Tools→Symbol Browser。符号浏览窗口中包含如下五个 tab 栏(如图 3.18 所示)：

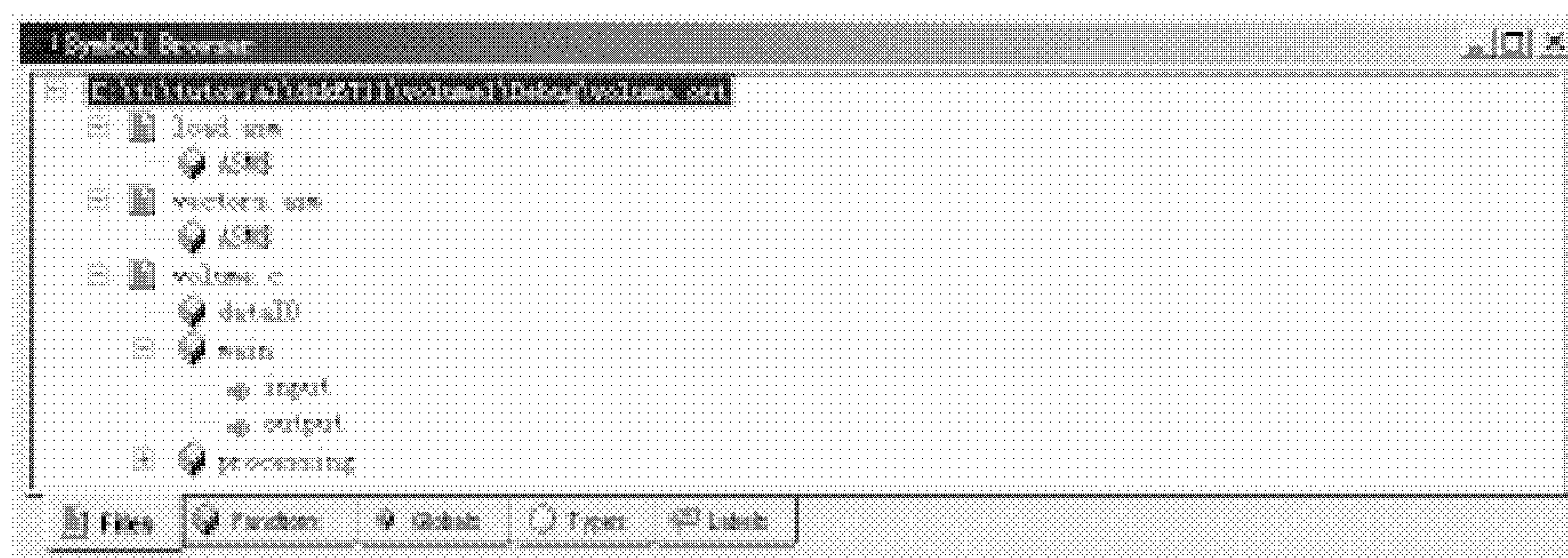


图 3.18　符号浏览窗口

• Files：显示当前已加载.out 文件中的所有源文件列表，而且每一源文件点开后会显示此源文件中的所有函数，进一步点开每一函数会显示此函数体中的所有局部变量。

• Functions：显示当前已加载的.out 文件中的所有函数列表，点开每一函数显示此函数体内的所有局部变量。

• Globals：显示当前已加载的.out 文件中的所有全局变量。

• Types：显示当前已加载的.out 文件中创建的所有类型定义，点开每一类型后可以看到其包含的原始数据类型。

• Labels：显示当前已加载的.out 文件中的所有标号。

双击符号浏览窗口中的文件或函数名，CCS 会自动在源代码编辑窗中打开此源文件或函数。

8) 图形显示窗口

CCS IDE 集成了先进的信号分析工具，这些工具能够使开发者观察整个信号的情况而不只是单个数值。为了使观察更直观、方便，CCS IDE 提供了对 DSP 中存储器数据的不同显示方式，包括时域/频域、图像、眼图等。

(1) 选择 View→Graph→Time/Frequency，打开图形属性设置窗口(Graph Property Dialog)，如图 3.19 所示。

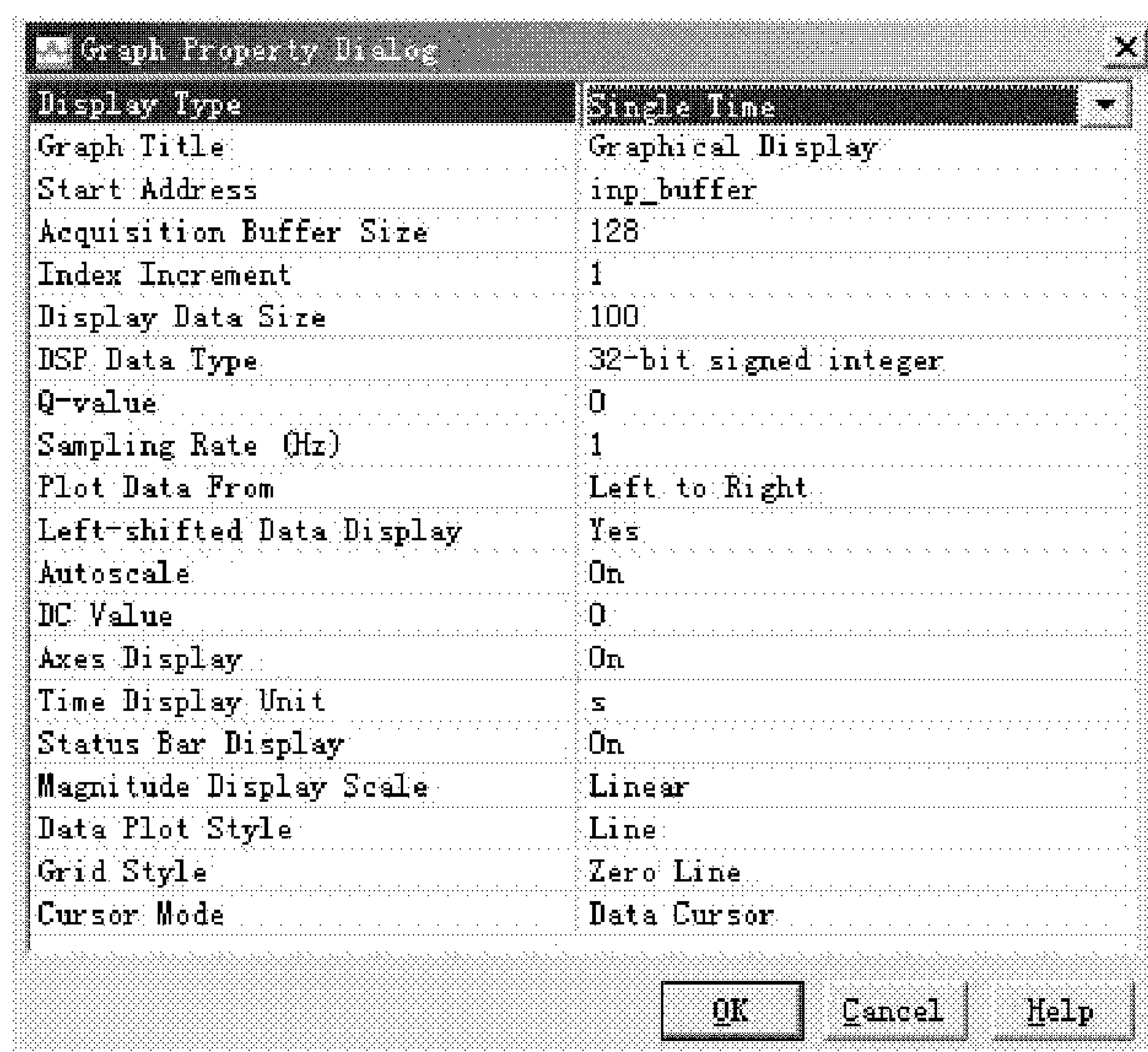


图 3.19　图形属性设置窗口

(2) 在图形属性设置窗口中设置图形显示的各种参数。

- Display Type：设置图形的显示类型，可以从其下拉菜单选择任一显示类型，包括：Single Time(单时间轴显示一段数据，对数据不作任何处理)、Dual Time(同时用两个时间轴显示两段数据，对数据不作任何处理)、FFT Magnitude(对数据作 FFT 处理，然后显示频域的幅度)、Complex FFT(对数据作 FFT 处理，同时显示频域的实部和虚部)、FFT Magnitude and Phase(对数据作 FFT 处理，同时显示频域的幅度和相位)、FFT Waterfall(对数据作 FFT 处理，同时利用多个帧数据的频域幅度形成 FFT 瀑布图形)、Constellation(利用二维星图显示输入信号的信息，输入信号分成 X、Y 两部分，每一部分信号沿自己的轴以其垂直轴为幅度轴画图)、Eye Diagram(利用眼图显示输入信号，即当输入信号的长度超过指定的显示长度，或输入/触发信号值的变化越过触发电平时，则接下来的输入信号折叠到指定的长度内与前面的输入信号叠加在一起进行显示。X 轴为指定的显示长度，Y 轴为信号的幅值)、Image(利用像素图像来显示数据，图像数据按 RGB 或 YUV 方式进行显示)。
- Graph Title：指定显示图形的标题。

以下的选项随 Display Type 的选择会有所不同。

- Interleaved Data Sources：此项有两种选择：Yes 或 No。对于需要两段或多段输入数据的显示类型(Dual Time、复数输入 FFT、Constellation、Eye Diagram、Image)，利用此项来指定这两段输入数据是从同一段存储空间间隔取值的(Interleaved Data Sources 为 Yes)，还是从不同的两段存储空间中取值的(Interleaved Data Sources 为 No)。
- Start Address：显示数据的首地址。
- Acquisition Buffer Size：指定主机数据缓冲区的长度。
- Index Increment：指定显示数据的采样间隔。
- Display Data Size：指定主机显示缓冲区长度。
- DSP Data Type：指定 DSP 数据类型。
- Q－Value：指定二进制小数点的位置。

- Minmum/Maximum：X/Y－Value：指定最小/最大X/Y的值。
- Autoscale：选择自动定标或指定最大显示范围。
- Sampling Rate(Hz)：指定数据的采样率，CCS用来计算显示时间和频率范围。
- Plot Data From：指定缓冲区数据的显示顺序。
- Left－shifted Data Display：此项控制数据缓冲区的数据如何放置在显示缓冲区中。如果Left－shifted Data Display为Yes，则显示缓冲区中的所有数据进行左移，数据缓冲区中的数据从最右端进入显示缓冲区；如果Left－shifted Data Display为No，则数据缓冲区中的数据会覆盖掉上次显示的缓冲区的数据，数据缓冲区中的数据也从最右端进入显示缓冲区。
- DC Value：此项设置显示信号的直流分量，即Y轴显示范围的中间点。
- Axes Display：此项指定是否显示坐标轴。
- Time/Frequency Display Unit：指定时间/频率显示单位。
- Status Bar Display：此项控制是否在图形窗口的底部显示状态栏信息。
- Magnitude Display Scale：设置幅度显示方法：线性显示或对数显示。
- Data Plot Style：选择数据的显示方法：连续线显示或栅栏显示。
- Grid Style：设置显示图形中的坐标网格。
- Cursor Mode：指定图形中游标的类型：没有游标(No Cursor)、坐标游标(Data Cursor，指定数据的坐标)和放大游标(Zoom Cursor)。

计算FFT时需指定的选项：

- Signal Type：指定输入FFT的信号为实数(Real)或复数(Complex)。
- FFT Framesize：指定每次FFT计算的信号样本数。
- FFT Order：指定FFT长度的2的幂次，即FFT长度$=2^{FFTorder}$。
- FFT Windowing Function：选择计算FFT时的窗函数。
- Number of Waterfall Frames：指定计算FFT Waterfall的帧数。
- Waterfall Height(%)：指定显示的Waterfall最大幅度占整个显示窗口的百分比。

星图显示类型指定的选项：

- Constellation Points：指定显示缓冲区的大小，显示缓冲区越大，则显示的信号历史就越长。如果数据缓冲区(acquisition buffer)中有新的数据到来，整个显示缓冲区中的数据会左移，新数据从右端进入显示缓冲区。
- Symbol Size：设置显示符号的大小。每一个星点都用一个'x'符号，此项设置'x'符号的大小。

眼图显示类型指定的选项：

- Trigger Source：指定是否需要触发源。如果选择Yes，则需要在Start Address－Trigger Source项中输入触发源的首地址。当触发源的变化越过触发电平时，接下来的源数据重新折回到显示窗的开始端(最左端)进行显示。
- Persistence Size：指定显示缓冲区的大小。数据缓冲区(acquisition buffer)中新的数据左移进显示缓冲区中。
- Display Length：设置显示信号的长度，而且当信号的长度超过Display Length的长度时，信号重新折回到显示窗的开始端进行显示。
- Minimum Interval Between Triggers：设置两个相邻触发点的最小采样间隔。

• Pre-Trigger(in samples)：此项控制信号的开始数据在显示窗口中的左右移动。如果为一正值，则开始点向右移动；如果为一负值，则开始点向左移动。

• Trigger Level：此值指定触发电平。

图像显示类型指定的选项：

• Color Space：此项指定图像数据解释及显示方式。有两种选项：RGB 或 YUV。

• Lines Per Display：此项指定图像的高度(单位：像素点)。

• Pixels Per Line：此项指定图像的宽度(单位：像素点)。

• Byte Packing to Fill 32 Bits：此项指定是否把数据打包成 32 位。

• Image Row 4 Byte Aligned：当 Byte Packing to Fill 32 Bits 选择 Yes 时，此项指定图像的每行数据是否以 4 字节为地址边界(即地址空间的最后两位为 0)。

• Image Origin：指定图像的原点在图形窗中的位置。

• Uniform Quantization to 256 Colors：选择是否把原始图像均匀量化成 256 色图像，量化后图像具有 8 级红色、8 级绿色和 4 级蓝色。如果此项选择 No，则图像按原始 RGB 图像进行显示。

• Error Diffusion：当 Uniform Quantization to 256 Colors 选择 Yes 时，此项指定是否分散量化误差。

• Bits Per Pixel：当 Interleaved Data Source 选择 Yes 时，此项指定每一像素的数据位数：8(3 位红、3 位绿、2 位蓝)、16(5 位红、6 位绿、5 位蓝)、24(8 位红、8 位绿、8 位蓝)和 32(最高字节不用，8 位红、8 位绿、8 位蓝)。

• Image RGB Order：当 Bits Per Pixel 选择为 16、24 或 32 时，此项指定 RGB 的顺序。

• Palette Option：当 Bits Per Pixel 选择为 8 时，此项指定把 8 位像素值转换为 8 位颜色值的模式。

• YUV Ratio：当 Color Spaec 为 YUV 时，此项指定 Y、U、V 采样之间的关系。

• Transformation of YUV Values：当 Color Space 为 YUV 时，此项指定把 YUV 转换成 RGB 格式。

4．Simulator Analysis 工具

Simulator Analysis 工具只能应用于 Simulator 调试平台。

Simulator Analysis 工具用来报告某一个或多个系统事件的发生情况，从而可以监视程序的执行过程。

按以下步骤来应用 Simulator Analysis 工具：

(1) 把可执行文件(.out)加载到 Simulator 中。

(2) 选择 Tools→Simulator Analysis，打开 Simulator Analysis 窗口，如图 3.20 所示。

C6000 Simulator Analysis

Event	Count/Status	Total Count	Break Address	Routine	Type
Program Cache Miss	Configured				Break
Program Cache Hit	0	0			Count
External Program Access	8	9728			Count
Program Fetch	1	1035			Count
Program Access Block 0	Triggered		0x3a44		Break
External Data Access	Configured				Break
Data Bank Conflict	Configured				Break
Data Access Block 0	Configured				Break

图 3.20　Simulator Analysis 窗口

此窗口包含如下部分：

- Event：事件名。
- Count/Status：计录两断点或步执行命令之间计数事件发生的次数。
- Total Count：记录自从开始分析事件或上次复位计数器后，事件发生的所有次数。
- Type：指示事件为断点还是计数事件。
- Break Address：断点事件发生时，程序计数器 PC 的当前地址。
- Routine：断点事件发生时，程序计数器 PC 当前所在的子程序名。

(3) 右击 Simulator Analysis 窗口，从弹出的菜单中选择 Analysis Setup，打开 Analysis Setup 对话框，此对话框中列出了 Simulator 支持的所有事件(events)。

Analysis Setup 对话框中列出的任何事件都可以配置成计数事件(Count Event)或断点事件(break Event)。

计数事件：记录程序运行过程中此事件的发生次数。

断点事件：在程序运行过程中，当此事件发生时会停止程序的执行。

在 Analysis Setup 对话框的事件列表中选中某一事件，在 Event 项中指定此事件类型(Count 或 Break)，选择 Enable analysis，然后点击 Add 按钮就把此事件添加到 Simulator Analysis 窗中。按上述方法可以把 Analysis Setup 对话框中的任何事件添加到 Simulator Analysis 窗中。如果清除 Enable analysis 项，就会禁止 Simulator Analysis 窗中的所有事件。如果删除某一事件，在 Analysis Setup 对话框中选择这一事件，然后点击 Delete 删除。

开发人员不但可以在 Simulator Analysis 窗中观察分析结果，而且还可以把分析结果输出到一个记录(log)文件中：点击 Analysis Setup 对话框中的 Open Log，在 Select Log File 对话框中指定记录文件名，点击 Open 退出 Select Log File 对话框。

(4) 点击 Close 关闭掉 Analysis Setup 对话框。

(5) 设置断点，运行程序，观察 Simulator Analysis 窗中的分析结果。

5．Emulator Analysis 工具

Emulator Analysis 工具类似于 Simulator Analysis 工具，但只能应用于硬件目标板平台，利用此工具来记录事件和设置硬件断点，监视程序的执行情况。

按以下步骤来应用 Emulator Analysis 工具：

(1) 选择 Tools→Emulator Analysis，打开 Emulator Analysis 窗口，此窗口类似于图 3.20。

(2) 右击 Emulator Analysis 窗口，从弹出的菜单中选择 Analysis Setup，打开 Analysis Events 对话框。

在 Analysis Events 对话框的 Count CPU Events 面板栏中利用单选按钮选择某一计数事件，并选择 Enable analysis events 项。

在 Analysis Events 对话框的 Set Up Hardware Breakpoints 面板栏中选择硬件断点事件和断点输出管脚，并选择 Enable analysis events 项。

(3) 点击 OK，把选择的事件添加到 Emulator Analysis 窗中，并退出 Analysis Events 对话框。

(4) 设置断点，运行程序，观察 Emulator Analysis 窗中的分析结果。

6．命令窗(Command Window)

在命令窗中用户可以直接输入 CCS 调试器命令。CCS 提供了多种调试命令，开发人员

可以直接用输入命令的方式来对程序进行调试，具体步骤是：

(1) 选择 Tools→Command Window，打开 TI Command Window 窗。

(2) 在 TI Command Window 窗的文本输入框中输入调试器命令。调试器命令总结于表 3.4 中，有些命令需带输入参数，如果要详细查看这些命令，在 Command 文本输入框中输入：help xxxx，就会显示此命令的在线帮助(xxxx 为命令名)。

表 3.4　调试器命令总结

命　令	功　能
?	计算表达式的值并显示其结果
abort, abortall	终止批处理文件的运行
alias	定义一个字符串来代表多个调试器命令
ba	添加一个软件断点
bd	删除一个软件断点
br	清除所有软件断点
cd,chdir	改变当前工作目录
cls	清除命令窗显示区中的所有信息
cnext	单步执行 C/C++程序的一条或多条语句，能够单步跳出调用的函数体
cstep	单步执行 C/C++程序的一条语句，进入调用的函数体
dir	显示某一目录中的所有内容
disp	向 Watch 窗中添加数组、结构体或指针变量
dlog	把命令窗显示区域的信息记录到文件中
e, eval	计算表达式的值
eb, event_break	把事件设置成断点模式
echo	在命令窗的显示区域中显示字符串
ecr, event_counter_reset	复位事件计数器
ecs, event_counter_start	把事件设置成计数模式
ed,event_disable	禁止事件
ee, event_enable	使能事件
el, event_list	列出事件及其状态
er, event_reset	复位事件
ext_addr	使能或禁止扩展存储器寻址
ext_addr_def	定义扩展存储器寻址范围
fc, fcx	比较两个文件
file	显示文本文件
files	显示当前打开的所有源文件列表
fill	用指定值填充存储器块
go	执行程序到指定地址处
halt	停止程序的执行
help	提供调试器命令帮助
if, else, endif	条件命令
ifpro	把目标 DSP 的设备号与指定的设备号进行比较
load	向 DSP 加载可执行代码
loop, endloop	执行循环命令

续表

命　令	功　　能
ma	指定一个新的存储器块
map	使能或禁止存储器映射
mc	把存储器地址映射到主机的某一文件
md	删除一个存储器块
mi	删除存储器地址与主机文件的连接
ml	列出存储器块信息
mr	删除调试存储器图中定义的所有存储器块
ms	把某一存储器块的内容保存到指定文件
next	单步执行下一条语句(C 或汇编)，单步跳出函数体
outputlines	设置命令窗中显示的最大行数
pause	暂停调试器的执行
pinc	把输入文件与中断管脚连接起来
pind	断开输入文件与中断管脚的连接
pinl	显示管脚信息
project	创建、编译链接 CCS 工程
prompt	改变命令行提示符
quit	退出 CCS IDE
realtime	使能实时模式
reconnect	初始化调试器与硬件仿真器之间的通信
reload	重新加载当前的可执行代码
reset	复位目标系统
restart, rest	复位程序计数器 PC 到程序的入口点
return, ret	返回到函数的调用处
run	运行程序代码
runb	执行一段指定的代码，并跟踪需要的 CPU 周期数
runf	清除所有断点，断开与仿真器的连接，从当前 PC 处开始高速运行
sconfig	加载显示配置
sload	加载符号表
sound	使能或禁止当出现错误时发生声响
ssave	保存当前显示配置到一个文件中
step	单步执行汇编指令或 C 语句，碰到函数调用时会进入函数体
stepon,stepoff	在执行批处理文件中的命令之前出现对话框提示
stopmode	使能停止模式
take	执行批处理文件中的命令
time	显示当前时间和日期
unalias	删除定义的批处理命令名
update	设置实时刷新模式
use	指定附加的目录
wa	在 Watch 窗中显示表达式的值
wd	从 Watch 窗中删除指定项
wr	关闭 Watch 窗

7．模拟外部中断(Pin Connect)

Pin Connect 工具只用于 Simulator 下，它允许用户在 Simulator 环境下对外部中断进行模拟。

按以下步骤来应用 Pin Connect：

(1) 创建一个指定中断间隔的数据文件，在此数据文件中指定中断间隔(以 CPU 时钟周期数为函数)，在每隔一指定的时钟周期数都会产生一个中断(从第 1 个时钟周期开始模拟)。数据文件中指定的中断间隔按如下格式给出：

clockcycle (rpt n|EOS)

• *clockcycle*：指定中断间隔的时钟周期数，可以用绝对时钟周期数或相对时钟周期数给出。

例　5　20　36　绝对时钟周期数，即在第 5、20、36 个时钟周期分别产生中断。

5　+5　20　相对时钟周期数，在第 5、10(即 5+5)、20 个时钟周期分别产生中断。

• rpt n|EOS：指定重复模式，如果选择 n，则重复 n 次；如果选择 EOS，则一直重复下去。

例　5(+10　+20) rpt　2：在第 5、15(即 5+10)、35(即 15+20)和 45(即 35+10)、65(即 45+20)个时钟周期分别产生中断。

10 (+5　+10) rpt　EOS：在第 10、15(即 10+5)、25(即 15+10)、30(即 25+5)、40(即 30+10)…个时钟周期分别产生中断，直到模拟结束。

(2) 选择 Tools→Pin Connect，打开 Pin Connect 窗口，如图 3.21 所示。外部中断管脚会在 Pin Name 栏中列出。



图 3.21　Pin Connect 窗口

(3) 把中断间隔数据文件与外部中断管脚进行连接。

双击某一中断管脚名或选择此管脚名，然后点击 Connect，在打开的 Open Pin File 对话框中指定此管脚的中断间隔数据文件。

断掉中断管脚与数据文件的连接：双击此管脚名或选择此管脚名，然后点击 Disconnect。

(4) 加载、运行可执行代码。

8．模拟端口数据传输(Port Connect)

Port Connect 工具只能用于 Simulator 下，它把一个主机文件连接到存储器地址(端口)上，可以从文件中读入数据或向此文件写入数据。

按以下步骤来应用 Port Connect：

(1) 选择 Tools→Port Connect，打开 Port Connect 窗，如图 3.22 所示。

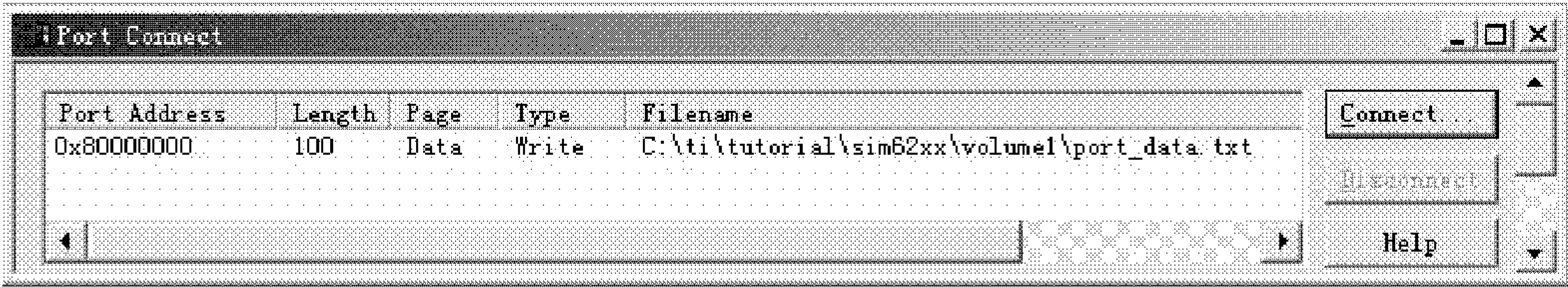


图 3.22　Port Connect 窗

(2) 点击 Connect，打开一个 Connect 对话框，在此对话框中指定连接选项。

在 Port Address 项中输入存储器地址，可以为绝对地址、C 表达式、C 函数名或汇编语言标号。

在 Length 项中指定存储器块的长度。

在 Page 项中选择存储器类型(只对于 C5000 系列 DSP)。

在 Type 项中选择 Write 或 Read。

(3) 点击 OK，关闭 Connect 对话框，同时打开 Open Port File 对话框，在此对话框中指定用来输入或输出数据的文件名。不同的存储器地址可以与同一个文件连接。

断掉某一存储器地址与数据文件的连接：在 Port Connect 窗中双击此端口地址或选择此端口，然后点击 Disconnect。

(4) 加载、运行可执行代码。

9．配置数/模、模/数转换器(Data Converter)

CCS IDE 中提供的 Data Converter 工具可以帮助用户快速、方便地对 DSP 外围的模/数、数/模转换器件进行软件开发。

Data Converter 的应用步骤为：

(1) 选择 Tools→ Data Converter Support，打开 Data Converter Support 对话框，如图 3.23 所示。

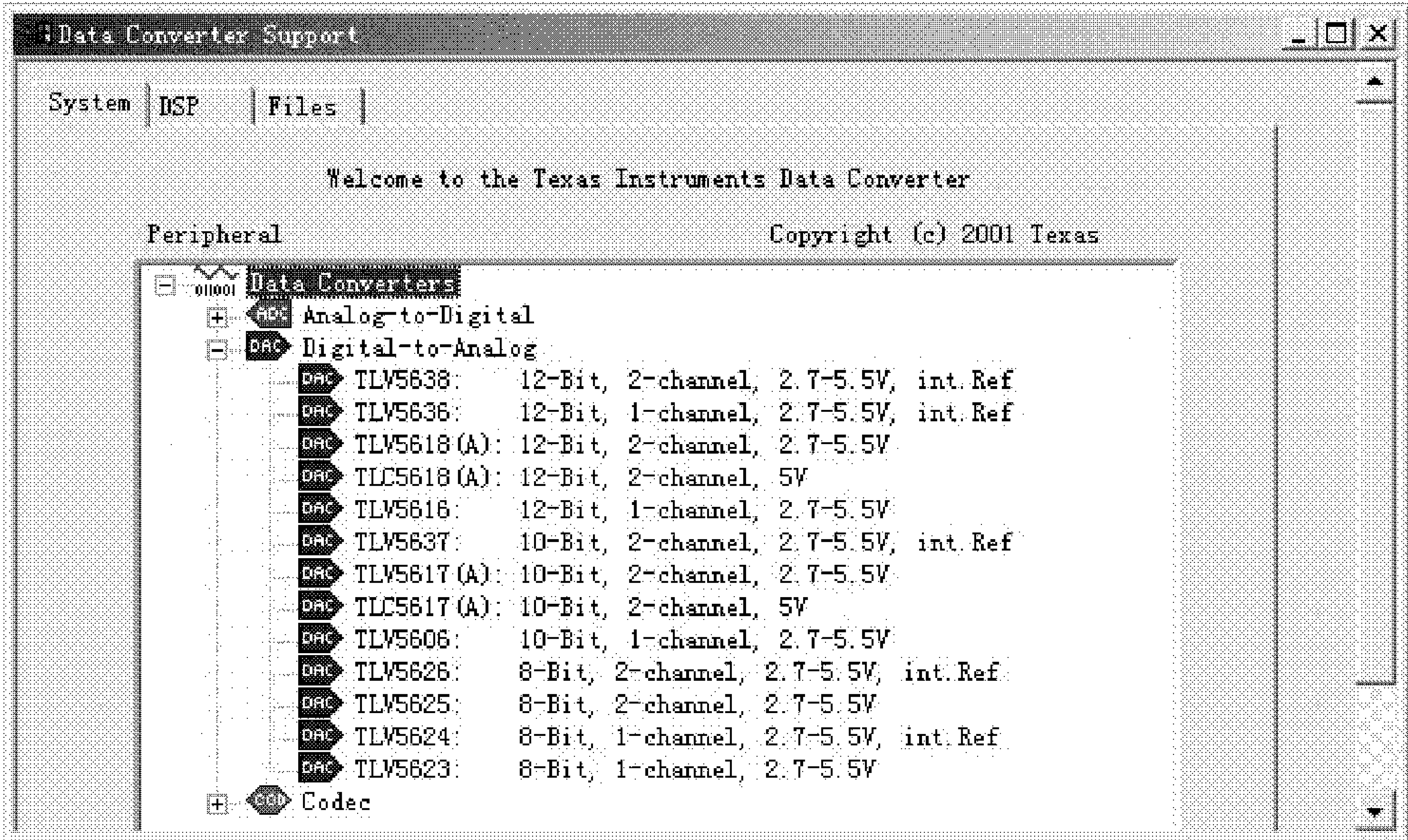


图 3.23　Data Converter Support 对话框

(2) 在 Data Converter Support 对话框的 System 面板中列出了 Data Converter 工具支持的所有模/数(ADC)、数/模(DAC)和编/解码(Codec)器件(点开每一项前面的＋号，就会显示出此项中包含的所有支持器件)。

右击某一数据转换器件，弹出的菜单中有三种选项：

- Add：向系统中添加此数据转换器件。
- Properties：选择此项会打开一个驱动文件配置对话框(Driver Files Configuration)：

  Input File1：设备驱动头文件的模板。

  Input File2：设备驱动源文件的模板。

  Output File1：Data Converter Support 工具为此设备生成的驱动头文件名。

  Output File2：Data Converter Support 工具为此设备生成的驱动源文件名。

  Create ISD File：只有选择此项才能生成上述输出文件(默认)。

  Add '.c' file to project：如果选择此项，会向工程中添加上述生成的输出文件(默认)。

  Driver Info 按钮：打开此设备的帮助文件。

  OK：退出 Driver File Configuration 对话框。
- Help：打开 Data Converter 工具的帮助文件。

(3) 向系统中添加用户指定的数据转换器件，右击 System 面板中的这一器件，从弹出的菜单中选择 Add。

(4) 打开 Data Converter Supprt 对话框的 DSP 面板栏，在 DSP 面板中配置 Data Converter 工具需要的 DSP 信息。

- DSP Type：从列表中选择 DSP 类型。
- DSP Speed：在文本输入框中输入 CPU 的时钟频率。
- CDB File：读入 DSP/BIOS 配置文件(.cdb)。如果工程中没有添加 DSP/BIOS 配置，此项为空。
- DSP On－Chip Peripherals：此项中列出了所选 DSP 器件的片内外围设备。
- Clear 按钮：清除此面板中的用户设置。
- Help 按钮：打开 Data Converter 工具的帮助文件。

(5) 配置所选的器件(ADC、DAC 或 Codec)。加入到系统中的每一个器件都会自动在 Data Converter Support 对话框中添加此器件的配置面板，在此面板中对所选器件进行配置。关于这些器件的配置信息，请查阅 CCS IDE 提供的帮助文件，本节不再一一介绍。

(6) 打开 Data Converter Support 对话框的 Files 面板，利用此面板生成配置文件。在此之前，用户应该已经在 CCS IDE 中打开了一个工程。

点击 Files 面板的 Write Files，开始代码产生并把代码自动添加到用户工程中。

如果选择 Files 面板的 Created files 项，会在产生代码后自动在 CCS IDE 的编辑窗中打开这些生成的文本代码。

(7) 编译链接、加载、运行工程，调试数据转换器是否工作正常。

10．事件触发 AET(Advanced Event Triggering)

事件触发工具只应用于硬件目标平台，不能应用于 Simulator。

事件触发工具包括两部分：事件分析工具和事件序列器。利用这些工具可以大大简化

硬件分析操作，提高硬件分析能力。

1) 事件分析(Event Analysis)工具

事件分析工具利用图形界面配置硬件调试任务(称为 jobs)。可按以下步骤来配置事件分析工具：

(1) 选择 Tools→Advanced Event Triggering→Event Analysis，打开 Event Analysis 对话框，如图 3.24 所示。

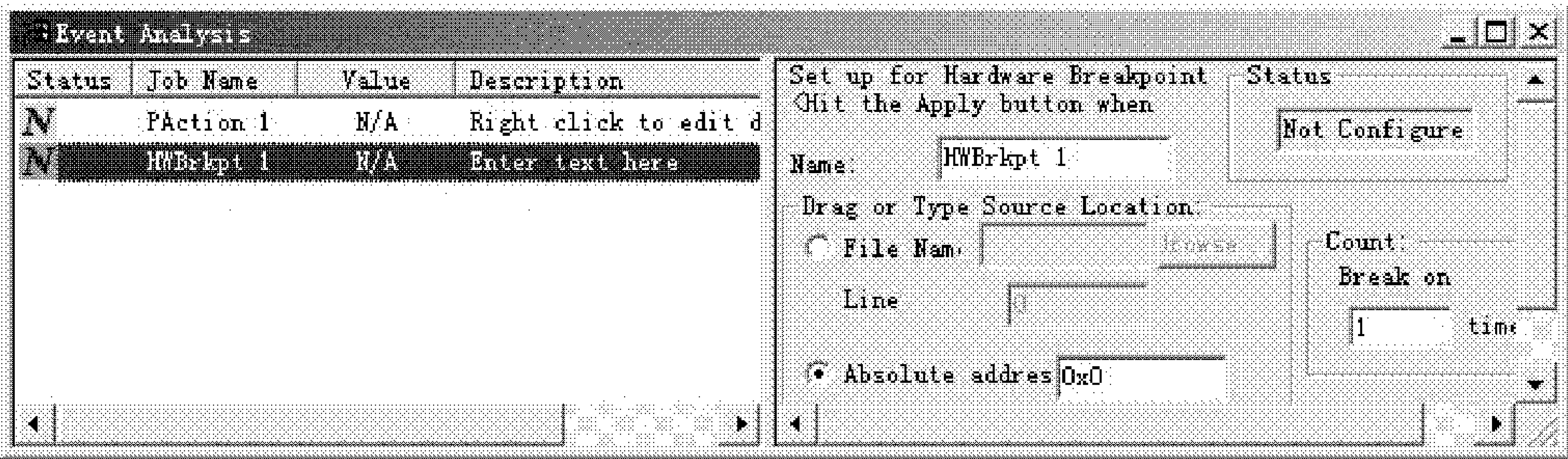


图 3.24 Event Analysis 对话框

Event Analysis 对话框的左半部分为添加的调试任务列表，包括状态、job 名、返回值、描述等，右半部分为调试任务的配置对话框。

(2) 向此事件分析窗中添加调试任务。

右击此窗，从弹出的菜单中选择需要添加的调试任务。事件分析工具可以配置如下调试任务：

硬件断点(Hardware Breakpoint)：在指定程序位置停止 CPU 的运行。

带计数的硬件断点(Hardware Breakpoint with Count)：当指定的硬件断点位置碰到的次数达到 n 次后才会停止 CPU 的运行，n 为指定的计数次数。

链式硬件断点(Chained Breakpoint)：只有先碰到指定的开始地址后才会在指定的结束地址处停止 CPU 的执行。

数据访问点(Data Actionpoint)：当指定的数据变量被访问时触发一事件发生。

程序访问点(Program Actionpoint)：当程序执行到某一指定位置时触发一事件发生。

硬件监视点(Hardware Watchpoint)：当指定的数据变量被访问时停止 CPU 的执行。

数据访问计数器(Data Access Counter)：记录对某一数据变量访问的次数。

一般计数器(Counter from Current Location，Counter in Range)：记录某一事件从当前 PC 位置开始的时钟周期数或发生的次数(Counter from Current Location)，或记录某一事件在指定代码范围内的时钟周期数或发生的次数(Counter in Range)。

看门狗定时器(Watchdog Timer)：当在指定代码范围内记录的时钟周期数超过一指定门限值时，Watchdog Timer 会触发一个信息对话框。

另一种添加调试任务的方法是：在 CCS IDE 的源代码编辑窗中打开需要添加的调试任务的源文件，高亮选中某一行(或一段)代码或数据变量，然后点击鼠标右键，从弹出的菜单中选择 Advanced Event Triggering→任务类型→任务名。

(3) 添加的调试任务必须进行正确配置才能使能。

在 Event Analysis 对话框的调试任务列表中选择某一添加的调试任务，Event Analysis

对话框的右半部分会自动出现此调试任务的配置对话框。对于不同类型的调试任务，此配置对话框的内容不同。开发人员可根据配置对话框上的提示很容易完成配置，或点击配置对话框上的 Help，打开此调试任务的配置帮助。完成配置后，点击 Apply，使能此调试任务。

右击 Event Analysis 对话框中的任一调试任务，从弹出的菜单中可以选择删除、禁止、使能此调试任务，或选择删除、禁止、使能此列表中的所有调试任务。

(4) 运行加载的可执行代码，观察调试结果。

2) 事件序列器(Event Sequencer)

利用 Event Sequencer 提供的工具，开发人员可以指定目标程序中的一些条件(即事件)，当这些条件发生时会自动启动指定的行为。Event Sequencer 提供了图形界面操作方法，开发人员可以方便、灵活地配置这些指定条件及其发生时的行为。

(1) 选择 Tools→Advanced Event Triggering→Event Sequencer，打开 Event Sequencer 窗口，如图 3.25 所示。

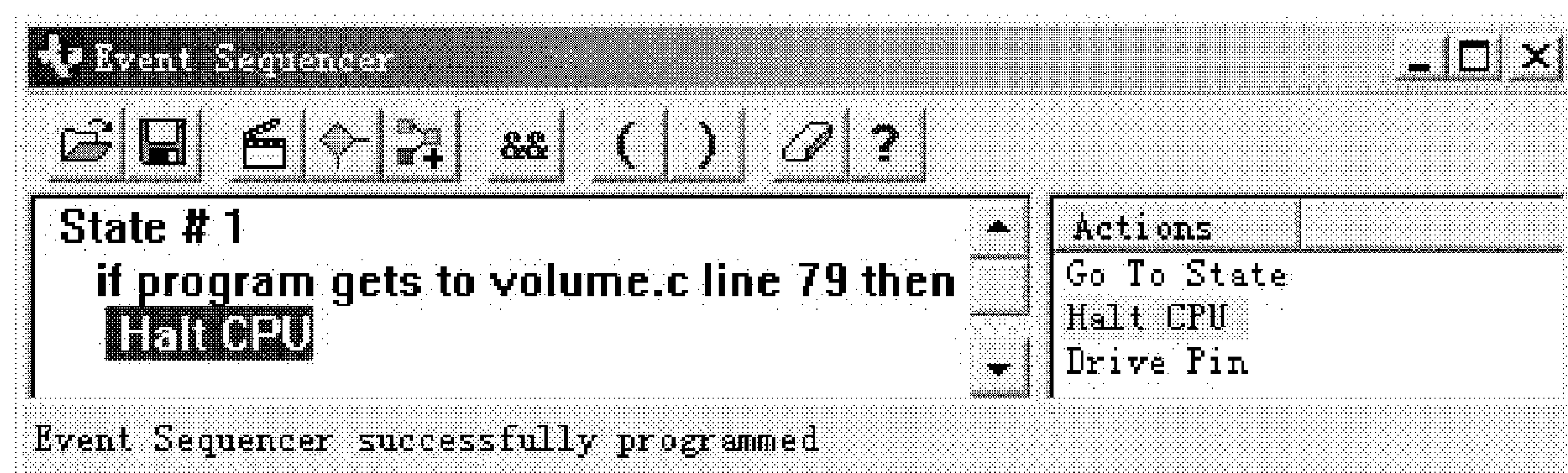


图 3.25　Event Sequencer 窗口

Event Sequencer 窗口的左半部分为创建的序列程序(Sequencer Program)，右半部分为指定条件发生时可供选择的行为。

(2) 创建序列程序(Sequencer Program)。

利用 Event Sequencer 工具栏中的工具或右击 Event Sequencer 窗，从弹出的菜单中可以选择添加一个全局行为或全局 If－Then 语句或新状态(state)。

在新添加的 If－Then 语句中，右击 If 后的 Undefined Event 项，从弹出的菜单中指定一个系统事件(System Event)或数据事件(Data Event)、程序事件(Program Event)、总线事件(Bus Event)等，也可以点击工具栏中的“与”操作符按钮，从而可以定义多个事件的“与”操作。右击 then 后的 Undefined Event 项，从弹出的菜单中选择 Define Action，或双击 Undefined Event 项，或选中 Undefined Event 项，然后双击 Event Sequencer 窗左边的 Actions 列表中的项，指定当 If 中的事件(条件)发生时启动的行为。

点击 Event Sequencer 工具栏中的擦除按钮，可擦除 Event Sequencer 窗中创建的所有序列程序(Sequencer Program)。也可以按以下操作禁止或使能这些序列程序：右击 Event Sequencer 窗口，从弹出的菜单中选择 Enable Sequencer Program(使能)或 Disable Sequencer Program(禁止)。也可以把这些序列程序保存到文件中：点击工具栏中的保存图标，指定文件名进行保存。

(3) 运行加载的可执行代码，观察调试结果。

11．统计代码的执行性能(Profiler)

利用 Profiler 工具可以统计一段代码的执行性能(即所占用的 CPU 时间等)，从而可以帮助开发人员优化出更加有效的代码，排除性能瓶颈问题。

(1) 选择 Profiler→Start New Session，开始新的性能统计任务，在 Profiler Session Name 对话框中输入此次性能统计任务的名称。

(2) 点击 OK，退出 Profile Session Name 对话框，同时打开一个统计结果观察窗口(MySession Profile)，如图 3.26 所示。

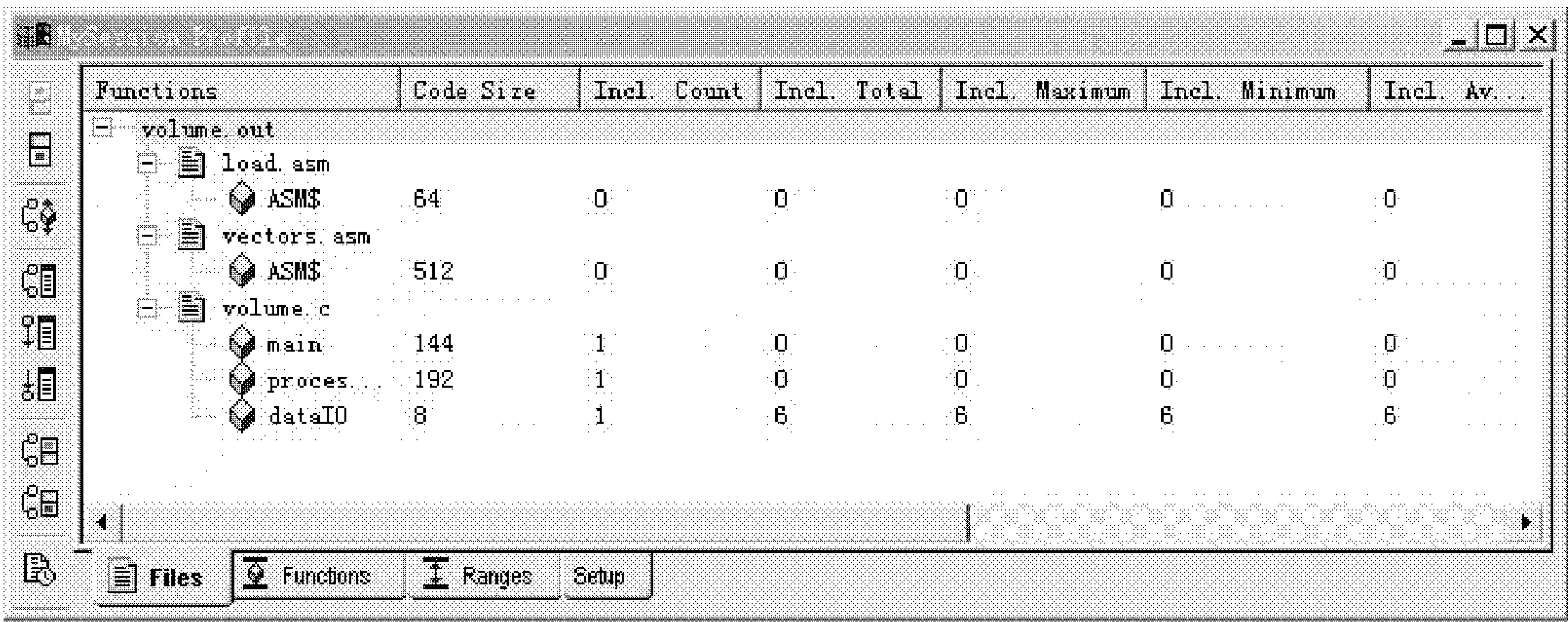


图 3.26　Profiler 统计结果观察窗口

Profiler 窗中包含如下四个子窗口：

• Files：显示所有包含被统计的函数或代码段的源文件，点击源文件前面的“+”号，可显示此源文件中包含的被统计的函数和代码段及其统计结果。

• Functions：显示所有被统计的函数及其统计结果。

• Ranges：显示所有被统计的代码段(不包括函数，被统计的函数在 Function 窗中显示)及其统计结果。

• Setup：显示统计执行的所有开始点和结束点。在程序执行过程中，当碰到结束点(end point)时统计过程结束，当碰到开始点(start point)时统计过程重新开始。

统计观察窗中显示的统计结果包含如下内容：

Code Size：被统计代码段的目标代码长度(即所占用的存储器单元的字节数)。

Incl.Count：给出此代码段被统计的次数。

Incl.Total：给出整个统计过程中执行此代码段所占用的所有指令周期数，其中包括执行此代码段中所有被调用子函数的指令周期数。

Incl.Maximum：给出统计过程中每次执行此代码段所花费的最大指令周期数，其中包括执行此代码段中所有被调用子函数的指令周期数。

Incl.Minimum：给出统计过程中每次执行此代码段所花费的最少指令周期数，其中包括执行代码段中所有被调用子函数的指令周期数。

Incl.Average：给出统计过程中多次执行此代码段的平均指令周期数，其中包括执行此代码段中所有被调用子函数的指令周期数。

Excl.Count：给出此代码段被统计的次数。

Excl.Total：给出统计过程中执行此代码段所花费的所有指令周期数，不包括执行此代码段中所有被调用子函数的指令周期数。

Excl.Maximum：给出统计过程中每次执行此代码段所花费的最大指令周期数，不包括执行此代码段中所有被调用子函数的指令周期数。

Excl.Minimum：给出统计过程中每次执行此代码段所花费的最小指令周期数，不包括执行此代码段中所有被调用子函数的指令周期数。

Excl.Average：给出统计过程中多次执行此代码段的平均指令周期数，不包括执行此代码段中所有被调用子函数的指令周期数。

(3) 向 Profile 窗中添加需要统计的代码段(包括函数)。

统计某一函数的执行性能时，可以用下述三种方法来向 Profile 窗中添加此函数：

① 在 CCS IDE 的源代码编辑窗口中打开包含此函数体的源文件，右击此函数体中的任一行源代码，从弹出的菜单中选择 Profile Function → in *MySession* Session(*MySession* 为用户在统计开始时输入的新任务名)，就会把此函数添加到 Profile 窗中。可以在 Functions 或 Files 窗口中查看此添加的函数。

② 在 CCS IDE 的源代码编辑窗口中打开包含此函数体的源文件，高亮选中此函数体中的任一行，然后用鼠标把此行拖到 Functions 窗口中，此函数就会添加到 Profile 窗中。

③ 点击 Profile 工具栏中的 Create Profile Area 图标，打开一个 Add Profile Area 对话框，如图 3.27 所示。在 Add Profile Area 对话框中设置如下参数：

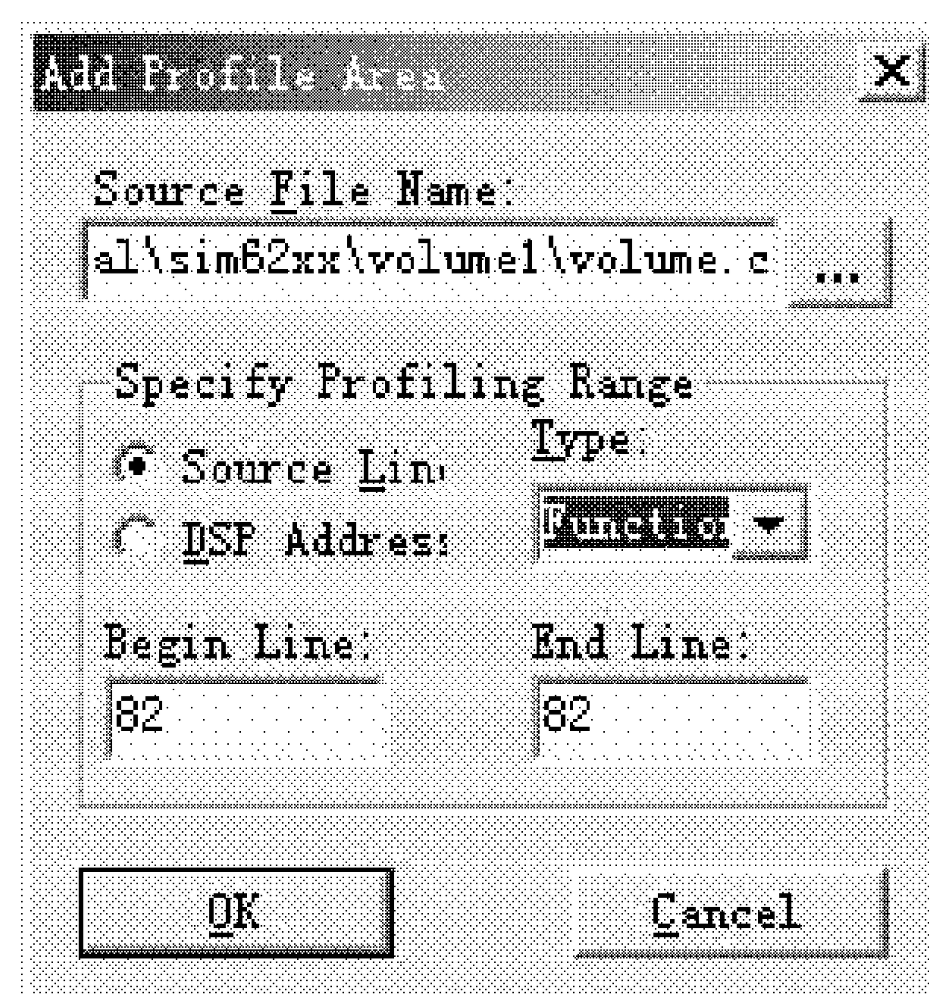


图 3.27　Add Profile Area 对话框

Source File Name：输入包含此函数体的源文件名及其路径。

Source Lines：利用单选按钮选择此项。

Type：从下拉菜单中选择 Function。

Begin Line：在此项中输入函数体中源代码的任一行号。

End Line：在此项中可以输入与 Begin Line 相同的值。

点击 OK 退出 Add Profile Area 对话框，同时此函数会添加到 Functions 窗中。

如果向 Profile 窗中添加所有的函数(对可执行代码中的所有函数进行性能统计)，则只需点击 Profile 工具栏中的 Profile All Functions 图标就可以了。

统计某一源代码段或反汇编代码段的执行性能，可以用下述三种方法来向 Profile 窗中添加此代码段：

① 在 CCS IDE 的源代码编辑窗中打开包含此代码段的源文件，高亮选中此代码段，然后右击，从弹出的菜单中选择 Profile Range → in *MySession* Session(*MySession* 为用户在统计开始时输入的新任务名)。可以在 Ranges 窗口中看到此代码段已被添加进来了。

② 在 CCS IDE 的源代码编辑窗中打开包含此代码段的源文件或反汇编窗，高亮选中此代码段，然后用鼠标把此代码段直接拖进 Ranges 窗口中。

③ 点击 Profile 工具栏中的 Create Profile Area 图标，打开 Add Profile Area 对话框，如图 3.27 所示。在 Add Profile Area 对话框中设置如下参数：

Source File Name：在此项中输入包含此代码段的源文件名及其路径。

Source Line：如果选择此项，则需要在 Begin Line 和 End Line 项中输入源文件的行号来指定统计代码段。

DSP Address：如果选择此项，则需要在 Begin Line 和 End Line 项中输入反汇编代码的地址来指定统计代码段。

Type：在下拉菜单中选择 Range。

Begin Line：如果前面选择了 Source Line，则在此项中输入统计代码段的起始行；如果前面选择了 DSP Address，则在此项中输入统计代码段的起始地址(反汇编窗)。

End Line：如果前面选择了 Source Line，则在此项中输入统计代码段的结束行；如果前面选择了 DSP Address，则在此项中输入统计代码段的结束地址(反汇编窗)。

如果需要删除某一函数或代码段，则在 Functions 或 Ranges 窗中选择此函数或代码段，然后按下键盘上的 Delete 键。

如果需要使能某一代码段(包括函数)，选择此代码段并点击 Profile 工具栏中的 Enable Profile Area 图标；如果要禁止某一代码段(包括函数)，选择此代码段并点击 Profile 工具栏中的 Disable Profile Area 图标。

(4) 设置执行统计的起始点和结束点。

当程序执行过程中碰到一个结束点时，会停止统计的执行。当碰到下一个开始点时，统计重新开始执行。

利用开始点和结束点可以从被统计代码中去除其中的一部分代码，即只统计其它部分的代码。可以按如下几种方法来创建统计的开始点和结束点：

① 在 CCS IDE 的源代码编辑窗中打开包含统计代码段的源文件或反汇编窗口，高亮选中指定为结束点或开始点的行，然后直接把此行拖进 Setup 窗的 Start Point 或 End Points 项中。

② 点击 Profile 工具栏中的 Create Setup End Point 图标来创建一个统计结束点，或点击 Create Setup Start Point 图标来创建一个统计开始点，打开一个 Add Setup Poin 对话框。在此对话框中设置如下参数：

Source File Name：在此项中输入包含统计代码段的源文件名及其路径。

Source Line：如果选择此项，则在下面的输入框中输入源文件的行号，来指定开始点或结束点。

DSP Address：如果选择此项，则在下面的输入框中输入反汇编窗中的指令地址，来指定开始点或结束点。

点击 Add，退出 Add Setup Point 对话框。

如果要删除某一开始点或结束点，则在 Setup 窗中选择此开始点或结束点，然后按下键盘上的 Delete 键；如果要删除所有开始点或结束点，则选中 Start Point 或 End Point 项，然后按 Delete 键。

(5) 使能统计时钟(Profile Clock)。

一般情况下，每当创建一个新统计任务时会自动使能统计时钟(Profile Clock)，如果 Profiler 菜单中的 Enable Clock 前有勾号，则表示统计时钟已使能，否则需要重新使能，即重新选择 Enable Clock。

(6) 设置统计时钟(Profile Clock)。

选择 Profiler → Clock Setup，打开 Clock Setup 对话框，如图 3.28 所示。在 Clock Setup 对话框中设置如下参数：

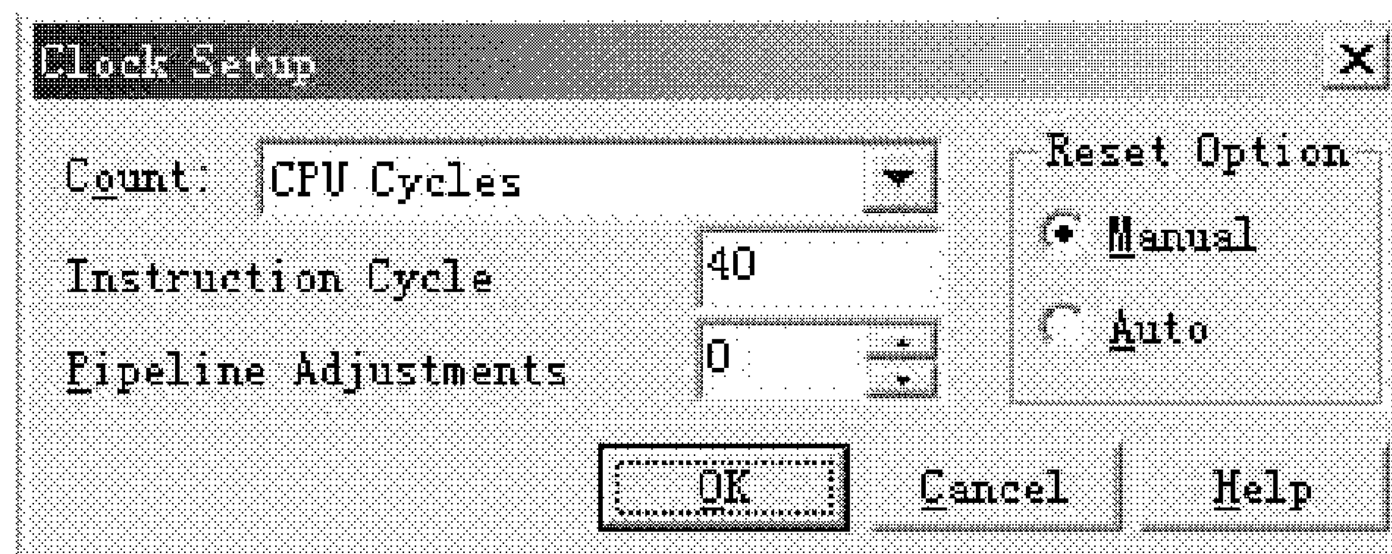


图 3.28　Clock Setup 对话框

Count：从其下拉菜单中选择需要统计的事件。所有目标调试平台都提供了 Cycle 事件，即对 CPU 时钟周期数进行统计，而有些调试平台还提供了对中断、子程序或中断返回、程序跳转、子程序调用等事件的统计。

Instruction Cycle Time：在此项中输入指令周期时间(即 CPU 执行每条指令所需的时间，单位为 ns)，用来把统计的时钟周期数转化成时间或频率进行显示。

Reset Option：选择时钟变量 CLK 的复位方法：Manual(手动)或 Auto(自动)。一般选择 Manual 模式。

Pipeline Adjustments：在此项中输入一个数字，用来去掉 DSP 清空流水线导致的附加时钟周期，以得到更精确的统计周期数。

(7) 运行加载的可执行代码，观察 Profile 窗中的统计结果。

清除代码段的统计信息(即对统计结果进行复位)：右击 Functions 或 Ranges 窗口，从弹出的菜单中选择 Clear Selected(清除所选择的某一代码段的统计结果)或 Clear All(清除所有代码段的统计结果)。

改变 Profile 窗中统计信息的显示模式：右击 Profile 窗，从弹出的菜单中选择 Property Page，打开 Profiler Control 属性对话框，在此对话框中选择信息显示模式：Cycle(时钟周期数)、Freq(频率)或 Time(时间)。

Profile 工具还可以把统计信息输出到一个文本文件(.txt)中：点击 Profile 工具栏的 Create Statistics Report 图标，打开一个 Create Report 对话框，在此对话框中指定保存的文件名及其路径，最后点击 OK 退出此对话框，统计信息就被保存到该文件中，同时会在 CCS IDE 的源代码编辑窗中打开此文本文件。

如果要禁止整个统计任务，则只需点击 Profile 工具栏中的 Turn Session Off 图标就可以了，再点击 Turn Session On 图标就会重新使能此统计任务。

如果要查看统计时钟的当前值(CLK 变量值)，选择 Profiler → View Clock，打开一个 Profile Clock 窗口，此窗口中显示当前时钟值。双击此窗口将把时钟值复位为 0。

3.3.2　代码调试演示例子

在 3.3.1 节中我们对 CCS IDE 提供的调试工具作了详细介绍，包括功能、操作方法和操作步骤等，本节再利用一个简单的例子来演示其中几个常用的调试工具，以帮助读者更快地掌握和使用常用的调试工具来调试自己编写的程序。

本节所用例子为 CCS 提供的演示例子，在 c:\ti\tutorial\dsk6711\volume1\目录下，我们先把整个目录内容拷贝到用户目录 c:\ti\myproject\volume1\中。volume1.pjt 工程中包含三个源文件：

(1) volume.c：C 语言主程序。

(2) load.asm：汇编语言文件，包含一个 load 子函数(在 volume.c 中对其调用)，此子函数只是利用一个循环 nop 来消耗 CPU 时间，不作任何操作。其用途在下面再进行演示。

(3) vectors.asm：汇编语言文件，此汇编文件定义中断服务程序，其中包含控制转向 C 程序入口点_c_int00 的 RESET(复位)中断服务指令包。

把 volume1.pjt 加载到 CCS IDE 中，双击工程视窗中的 volume.c 文件，会在 CCS IDE 的源代码编辑窗中打开此文件。volume.c 的源代码如下所示：

```
#include <stdio.h>
#include "volume.h"
 /* Global declarations */
int inp_buffer[BUFSIZE];           /* processing data buffers */
int out_buffer[BUFSIZE];
int gain = MINGAIN;                /* volume control variable */
unsigned int processingLoad = BASELOAD;   /* processing routine load value */
struct PARMS str ={2934, 9432, 213, 9432, &str};
 /* Functions */
extern void load(unsigned int loadValue);
static int processing(int *input, int *output);
static void dataIO(void);
        /* ======== main ========*/
  void main( )
   {
       int *input = &inp_buffer[0];
       int *output = &out_buffer[0];
       puts("volume example started\n");
         /* loop forever */
       while(TRUE)
         {
 /* Read input data using a probe-point connected to a host file.
    Write output data to a graph connected through a probe-point    */
              dataIO( );
             #ifdef FILEIO
              puts("begin processing")            /* deliberate syntax error */
             #endif
                /* apply gain */
              processing(input, output);
            }
        }

/* ======== processing ======== */
  static int processing(int *input, int *output)
   {
       int size = BUFSIZE;
```

```
            while(size- - ){
                        *output++ = *input++ * gain;
                       }
             /* additional processing load */
             load(processingLoad);
             return(TRUE);
        }
    /* ======== dataIO ======== */
        static void dataIO( )
         {
            /* do data I/O */
            return;
          }
```

在 volume.c 中，主函数打印一行信息后，进入一个无限循环体中，在此循环体内，调用 dataIO 和 processing 函数，其中 processing 函数对输入缓冲区内的每一个值乘上一增益(Gain)后，再将结果存储到输出缓冲区内。在此例子中，采用 CCS 的探点(Probepoint)工具将主机上的数据文件写入到输入缓冲区 inp_buffer 中，故 dataIO 执行空操作。

1．编译链接工程、加载可执行代码

选择 Project→Rebuild All 或点击工具栏中的重新编译链接图标，开始编译链接 volume1.pjt 工程，生成可执行代码 volume1.out。

选择 File→Load Program，把可执行代码 volume1.out 加载到目标 DSP 中。

2．利用 Watch 窗观察变量的值

选择 Debug→Go main，运行到主程序 main 的入口点，即初始化全局变量。选择 View→Watch Window 或点击工具栏中的 Watch Window 图标，打开 Watch Window 窗口。在 Watch Window 窗的 Watch1 子窗中输入变量 str，会显示出+str={…}，“+”表示该变量为结构体变量或数组。点开 str 左侧的“+”，显示出 str 中的值。再在 Watch1 窗中输入另一变量 gain，显示其值。Watch Window 窗中的显示结果参看图 3.16。

还可以直接点击工具栏中的 Quick Watch 图标，在 Quick Watch 对话框中输入要查看的变量。更为简单的方法是在 CCS IDE 的源代码编辑窗中打开 volumn.c，高亮选中需要查看的变量，然后右击鼠标，从弹出的菜单中选择 Add to Watch Window 或 Quick Watch 就可以了。

3．代码执行时间统计(Profile)

(1) 选择 Profile→Start New Session，打开一个新的统计任务窗口。下面演示统计代码：puts(”volume example started\n”); 的执行时间。

(2) 在 volume.c 的编辑窗口中高亮选中 puts(”volume example started\n”); 这一语句，然后右击鼠标从弹出的菜单中选择 Profile Rang→in MySession Session，把此行添加到 Profile 窗中。打开 Ranges 子窗口，可以观察到添加的统计代码。

(3) 在 While(TRUE)行处设置一断点。

(4) 选择 Profiler→Enable Clock，使能时钟。

(5) 运行程序，在 Profile 窗中观察统计结果，如图 3.29 所示。

图 3.29　Profile 窗中的统计结果

4．探点(Probepoint)设置及数据读取

探点在 DSP 程序调试时是非常有用的工具，利用探点可以实现主机文件向目标 DSP 输入数据，或目标 DSP 向主机文件输出数据，还可以利用新数据对窗口(如图形窗口)进行更新。同断点一样，探点也能中断目标 DSP 的运行，但探点只是暂时中断程序执行，当完成探点任务后可使目标 DSP 从当前 PC 位置继续开始执行。

(1) 在 volume.c 源代码的 dataIO( )；行处设置一探点。

设置探点的方法为：点击工具栏上的探点设置图标，或右击鼠标从弹出的菜单中选择 Toggle Probe Point。

(2) 选择 File→File I/O，打开 File I/O 对话框，在 File I/O 对话框中设置输入文件与探点的连接。

点击 File Input 面板上的 Add File，打开 c:\ti\myprojects\volume1\sine.dat 文件。在 Address 项中输入 inp_buffer，Length 项中输入 100，选择 Wrap Around，然后点击 Add Probe Point，打开 Probe Points 面板，在 Probe Points 探点列表中选择前面已设置的探点(在 dataIO( );行处)，在 Connect 项的下拉菜单中选择刚刚添加的数据文件 sine.dat，点击 Replace 把探点与数据文件连接。此时可以看到，Probe Point 列表中的此探点已与数据文件进行了连接。点击 OK 退出 Break/Probe Points 对话框。此时，可以观察到 File I/O 对话框的 Probe 项变成了 Connected，表示探点与数据文件已完成了连接。点击 OK 退出 File I/O 对话框，同时打开一个文件 I/O 控制工具条，用来监视和控制文件数据的输入和输出进程。

(3) 清除掉所有断点，运行程序。

程序运行后，可以从文件 I/O 控制工具条中观察到数据从文件向目标 DSP 循环输入数据的过程。

5．对目标 DSP 中的数据进行图形显示

为了测试更直观、方便，CCS IDE 对目标 DSP 的存储器数据提供了多种不同的显示方式，包括时域/频域、图像、星图、眼图等，详细使用方法请查阅 3.3.1 节中的介绍。下面利用这种数据显示功能对本例中的输入/输出缓冲区中的数据进行显示(包括静态和动态显示)。

1) 静态图形显示

选择 View→Graph→Time/Frequency，打开 Graph Property Dialog(图形属性窗口)。在图形属性窗口内，设置显示类型、图形标题、起始地址、数据缓冲区长度、显示缓冲区长度、DSP 数据类型、自动定标、最大 Y 值等诸选项，这些选项的详细定义请查阅 3.3.1 节。本例中，图形属性窗口的设置如图 3.30 所示。

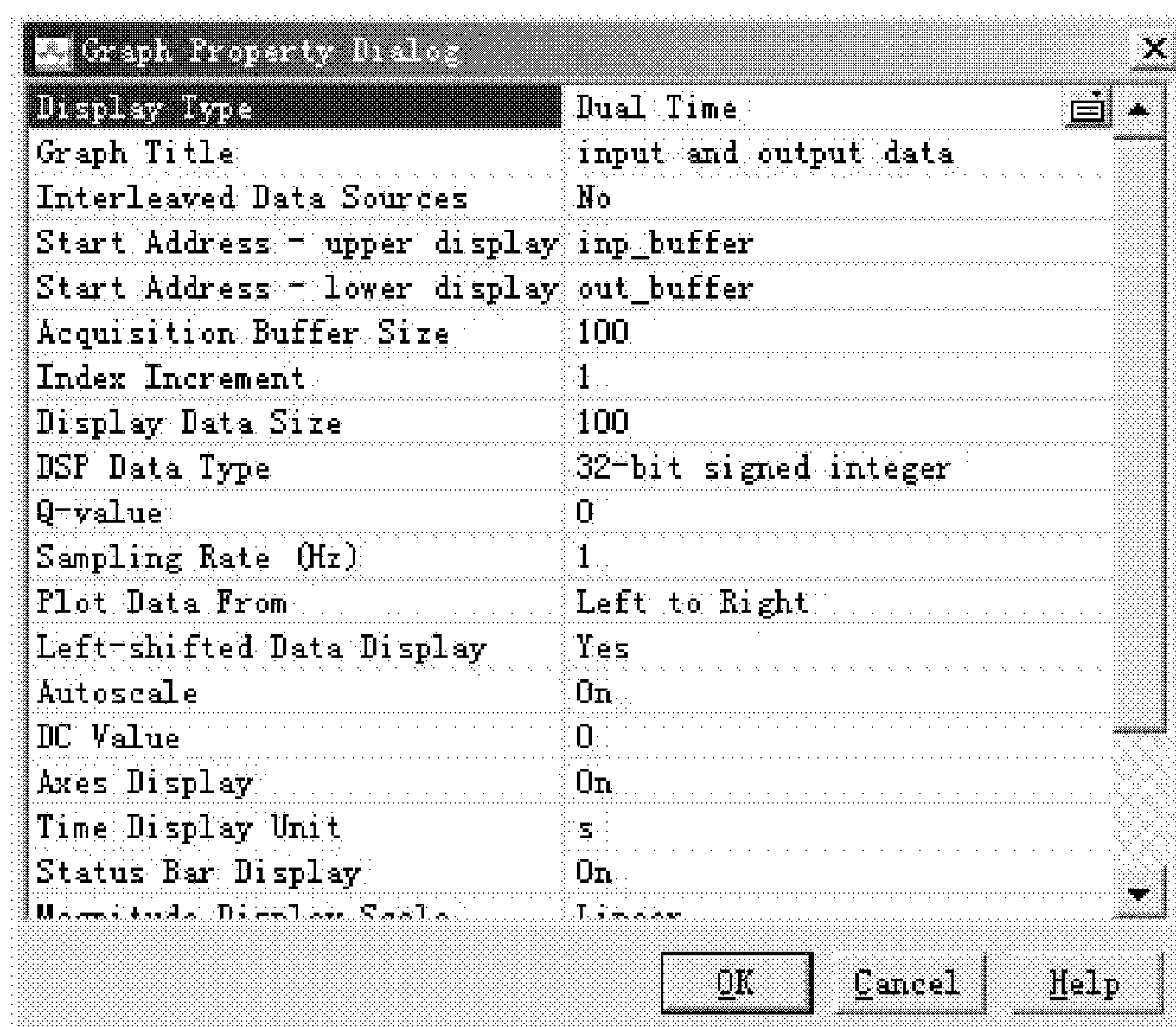


图 3.30　图形属性窗口的设置

点击 OK，退出图形属性窗口，同时打开数据显示窗口，如图 3.31 所示。图 3.31 的上半部分曲线为输入数据(inp_buffer 缓冲区中的数据)显示结果，下半部分曲线为输出数据(out_buffer 缓冲区中的数据)显示结果，即对输入数据乘上一增益(gain)后的结果，MINGAIN 在 volume.h 头文件中定义为 2，因此输出数据相当于对输入数据放大 2 倍，从图 3.31 中可以看到这一结果。

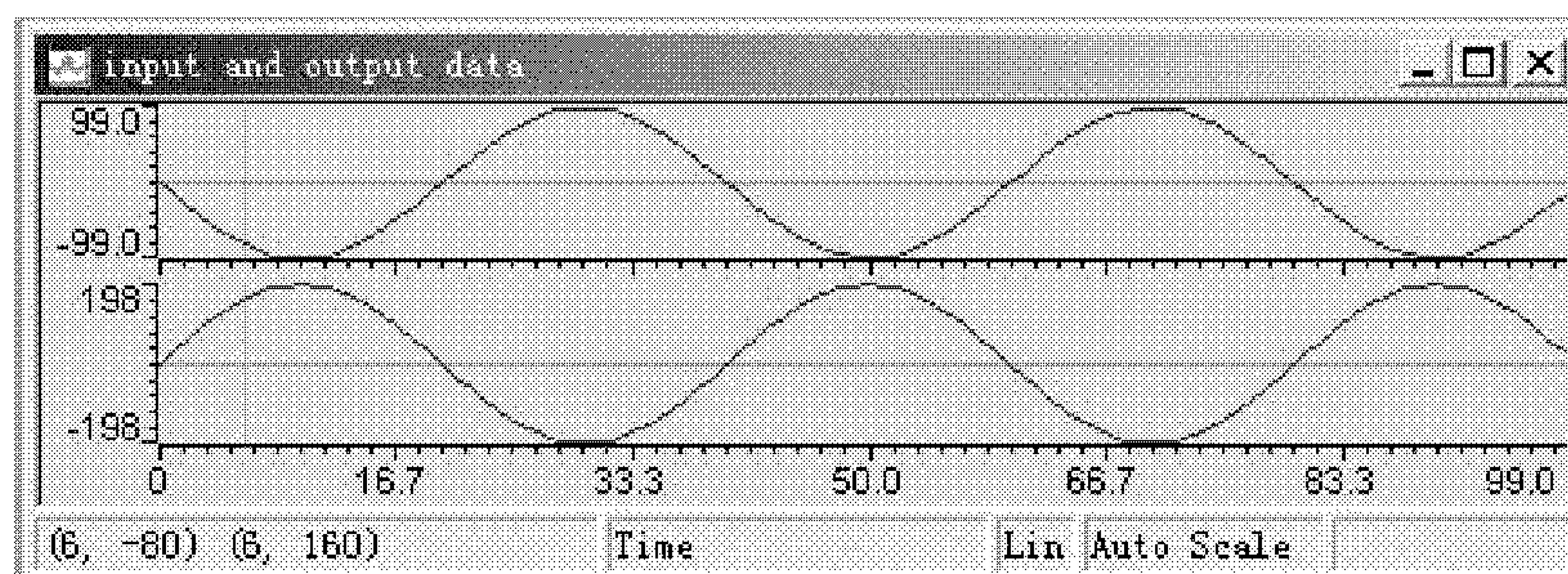


图 3.31　输入/输出缓冲区的数据显示结果

2) 动态图形显示

为了使开发者能更清楚地观察处理结果的变化情况，要求显示的图形能随处理的进行而动态更新。我们在前面提到过，探点(Probepoint)不但可以用来在主机文件和目标 DSP 之间传递数据，而且还可以用来更新某一窗口，探点只是暂时地停止程序的执行，当完成数据传递或更新窗口后继续执行程序，因此利用探点可以完成图形的动态显示。

在 volume.c 文件的 dataIO( );行处再添加一个探点，此探点不是用来进行主机文件与目标 DSP 之间的数据传递，而是用来更新前面的图形显示窗口。选择 Debug→Probe Points，打开 Break/Probe Points 对话框的 Probe Points 面板，在 Probe type 项中选择 Probe at Location，在 Location 项中输入 volume.c line ××(××为 dataIO( );所在的行号，与第一个探点的行号相同)，在 Connect 项中选择前面已打开的图形窗口，然后点击 Add 把此探点添加到下方的 Probe

Point 列表中，在 Probe Point 列表中可以看到当前已有两个探点，它们在源文件 volume.c 的同一行，其中一个探点与 c:\ti\myprojects\volume1\sine.dat 文件相连接，而另一个探点与图形窗 input and output data(在前面的图形属性窗中设置的标题名)相连接，最后点击 OK 退出 Break/Probe Points 对话框。在目标程序运行过程中可以看到动态更新的输入/输出数据显示窗口。

当然我们也可采用另一种方法来动态更新窗口：在 volume.c 的 dataIO( );行处再设置一断点，这时此行分别被设置了断点和探点，探点用来从主机文件向目标 DSP 输入数据，断点用来更新窗口；然后选择 Debug → Animate 或点击工具栏中的动态运行图标，对程序实现动态执行。程序的动态执行可以看作是：运行——停止(当遇到断点时对窗口进行更新)——继续运行过程。这样，也可以在输入/输出数据显示窗口上看到动态更新的图形。

实际上，程序运行结果将随程序中参数的变化而变化。在本例中，输出数据是输入数据与增益 gain 相乘的结果，改变 gain 的值，输出信号的幅度会相应变化。一种改变 gain 的方法是在 Watch 窗中加入 gain 变量，在此窗内对其值进行修改，gain 值的改变可直接从输出波形中反映出来。另一种改变 gain 的方法是采用 GEL 语言，按如下步骤来实现：首先选择 File → Load GEL，在 Load GEL 对话框内选择 volume.gel 并打开；再选择 GEL → Application Control → Gain，这时会打开一个标题为 gain 的控制工具条，用鼠标可以方便地移动游标从而实现对 gain 值的改变。

6．结合 MATLAB 来调试 CCS 中的程序

MATLAB 具有强大的分析和可视化功能，使用起来非常方便、灵活、简单，对于工程和算法研究人员来说，MATLAB 是一种必不可少的工具。通过 MATLAB 与 CCS 的连接，可实现在 MATLAB 环境下对目标 DSP 的存储器或寄存器数据进行访问，再利用 MATLAB 强大的分析和可视化工具对其数据进行分析和可视化。从 MATLAB 6.0 开始提供了这种 MATLAB 与 CCS 的连接工具，利用此连接工具，在 MATLAB 环境下就可以实现对目标 DSP 中数据的访问，而且也可以实现对工程的编译、链接、加载运行、设置断点和探点等功能。在本书的第 5 章中对此连接工具进行了详细介绍。本节演示如何利用此连接工具，在 MATLAB 环境下修改目标程序中某个变量的值、读目标 DSP 中的数据处理结果，并利用 MATLAB 对数据进行显示。需要说明的是，不可能只在 MATLAB 下就能够完成对 CCS 的所有操作，MATLAB 和 CCS 对某些调试任务各有方便之处，它们应该配合使用，即它们都应该在桌面上打开，用户经常需要在两者之间进行切换。作者强烈推荐用户应该把 MATLAB 和 CCS 配合使用，即利用 MATLAB 读取处理结果并进行分析、修改 DSP 中的数据，再利用 CCS 进行其它的零碎操作，包括设置 DSP/BIOS 来进行实时性调试等。

结合 MATLAB 调试 CCS 的具体步骤是：

(1) 打开 MATLAB6.5 的命令窗并查看 CCS 中安装的目标板信息。

在 MATLAB 命令窗中输入 ccsboardinfo，查看 CCS 中安装的目标板信息，显示如下。

```
        ccsboardinfo
 Board  Board                              Proc  Processor                 Processor
 Num    Name                               Num   Name                      Type
 ---  ---------------------------------- ------- ------------------------  ------------
  1   C6xxx Simulator (Texas Instrum ...    0    CPU_1                     TMS320C6701
  0   C6x11 DSK (Texas Instruments)         0    CPU_1                     TMS320C6x1x
```

由于主机系统上安装了一块 C6711 DSK 目标板和一个 C6701 Simulator 软件模拟器并且已在 CCS 中配置好，因此利用 ccsboardinfo 命令来查看 CCS 中安装的目标板信息时，就会显示上述两块目标板的信息，其中 0 号目标板为 C6711 DSK 硬件板，1 号目标板为 C6701 Simulator 软件模拟器。每块板上只有一个 DSP，因此 DSP 号都为 0。板号和 DSP 号是 CCS 自动给安装在主机系统上的目标板及其 DSP 分配的，用户不能进行修改。

(2) 创建 MATLAB 和 CCS 中的 TI C6711 DSK 目标板的连接。

```
cc=ccsdsp ('boardnum', 0, 'procnum', 0);
```

上述函数创建了 MATLAB 的一个对象 cc，此对象用来与 C6711 DSK 目标板上的 DSP 进行连接。利用连接对象 cc，就可以对此 DSP 进行访问。

(3) 读目标 DSP 中的数据并对其画图，如图 3.32 所示。

```
inp_buffer=createobj(cc, 'inp_buffer');      %创建目标程序中全局变量 inp_buffer 的嵌入式对象
out_buffer=createobj(cc, 'out_buffer');      %创建目标程序中全局变量 out_buffer 的嵌入式对象
inpbuffer=read(inp_buffer);                  %读目标 DSP 中 inp_buffer 数组的所有值，并把它放到
                                             %MATLAB 的 inpbuffer 变量中
outbuffer=read(out_buffer);                  %读目标 DSP 中 out_buffer 数组的所有值，并把它放到
                                             %MATLAB 的 outbuffer 变量中
plot(inpbuffer); hold on; plot(outbuffer);       %对输入和输出数据画图
```

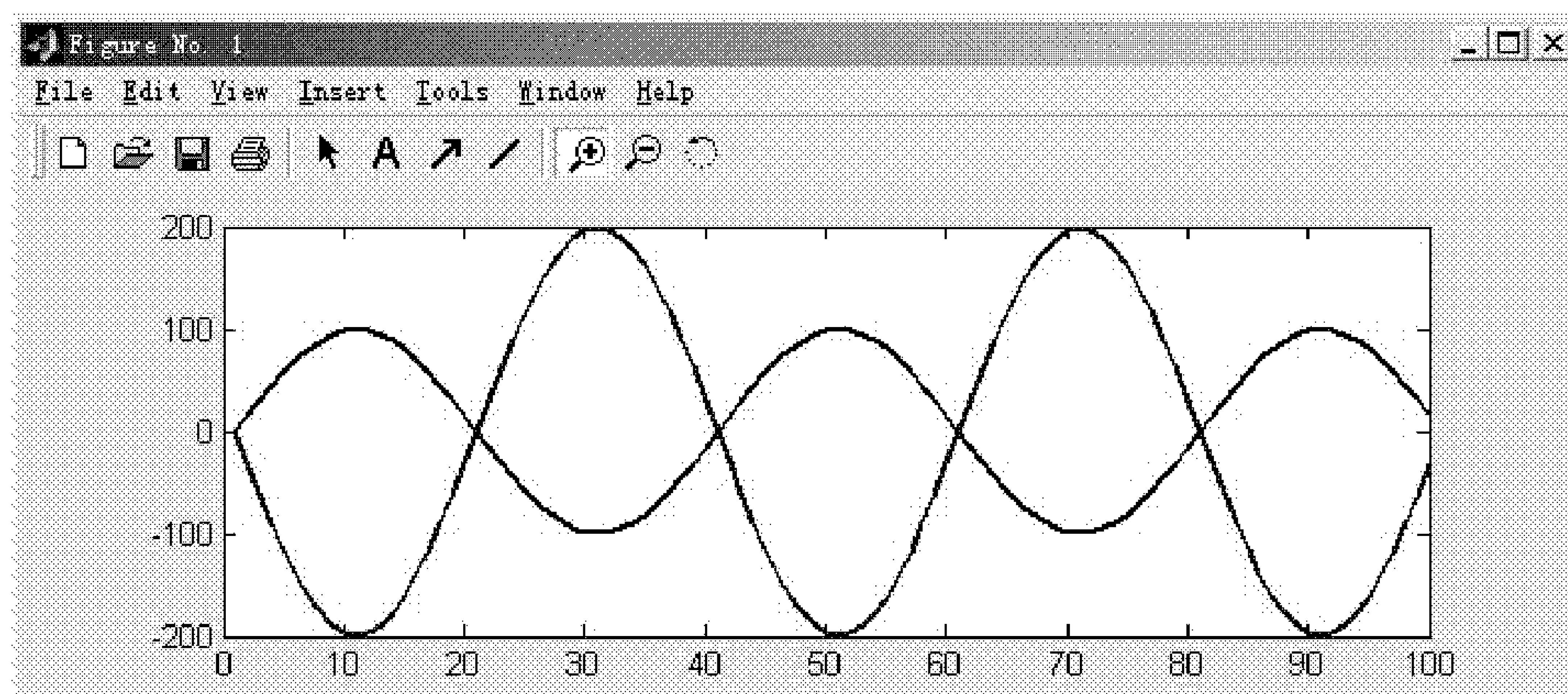


图 3.32　利用 MATLAB 对目标 DSP 中数据进行画图显示

(4) 修改目标 DSP 中的数据。

```
write(inp_buffer,[1:100]);      %修改目标 DSP 中 inp_buffer 的值
```

再利用 CCS 中的图形窗画出 inp_buffer 修改后的值。由此看到，利用 MATLAB 来修改目标 DSP 的数组非常方便、简单，这是 CCS 中的调试工具很难完成的。利用探点或直接加载数据，也可以修改数组，但需要首先产生数据文件，而在 MATLAB 下可以直接修改目标 DSP 中的变量值，而且 MATLAB 提供的数据图形分析功能远远超过 CCS。

在前面的例子中，CCS 对输入/输出数据显示窗口进行了动态更新，下面再利用 MATLAB 来修改变量 gain 的值，从而可以在动态数据显示窗口中观察输出结果的变化。

```
gain=createobj(cc, 'gain');     %创建 gain 的嵌入式对象
write(gain,1);   %修改 gain 的值为 1，并在 CCS IDE 的图形窗口中观察输出结果的变化
```

```
write(gain,2);     %修改 gain 的值为 2，并在 CCS IDE 的图形窗口中观察输出结果的变化
    ⋮              %任意修改 gain 的值
```

(5) 利用 MATLAB 生成一个可加载到目标 DSP 中的数据文件。

利用探点或选择 File→Data→Load 指定文件名、开始地址和长度，把数据文件中的数据加载到目标 DSP 中，此数据文件具有特定的格式。下面的一段 MATLAB 程序可以用来产生此数据文件(对于 C6000DSP)，包括十六进制格式、整型格式、长整型格式、浮点数格式。

```
fs=1000; f1=100; f2=400; f3=300;
x=100*sin(2*pi*f1*[1:200]/fs)+sin(2*pi*f2*[1:200]/fs)+sin(2*pi*f3*[1:200]/fs);   %产生数据
d=2;  %选择数据类型：1=hexadecimal, 2=integer, 3=long, 4=float
fid=fopen('sine.dat', 'w');                     %打开数据文件
fprintf(fid,'1651 %d 0 0 0\n', d);              %输出文件头
switch d
    case 1
        fprintf(fid, '0x%x\n', x);              %输出 32 bit 十六进制格式
    case 2
        fprintf(fid, '%d\n', round(x));         %输出 32 bit 整型格式
    case 3
        fprintf(fid,'%12.1f\n', round(x));      %输出 40 bit 长整型格式
    case 4
        fprintf(fid, '%f\n', x);                %输出 32 bit 浮点格式
  end
fclose(fid);
```

(6) 把 CCS 输出的数据文件(文本文件)读入到 MATLAB 中进行分析处理。

利用探点或选择 File→Data→Save 指定文件名、开始地址和长度，把数据保存到主机文件中，接下来再把此数据文件读入到 MATLAB 中进行分析处理。下面的一段 MATLAB 程序用来读取此数据文件。

```
x=zeros(100, 1);                     %定义数据距阵
Len=length(x);
fid=fopen('sine.dat', 'r');          %打开数据文件
head=fscanf(fid, '%d\n', 2);         %输出文件头
fseek(fid, 20, - 1);
switch head(2)
  case 1
      x=fscanf(fid, '%x\n', Len);    %读入十六进制格式
  case 2
      x=fscanf(fid, '%d\n', Len);    %读入 32 bit 整型格式
  case 3
      x=fscanf(fid, '%f\n', Len);    %读入 40 bit 长整型格式
  case 4
      x=fscanf(fid, '%f\n', Len);    %读入 32 bit 浮点格式
end
fclose(fid);
```

(7) 利用 MATLAB 中的 FDATool 工具来分析、设计滤波器，并把滤波器系数输出到目标 DSP 中。

MATLAB 中的 FDATool 工具是信号处理人员经常使用的工具，FDATool 利用图形界面来设计和分析滤波器，通过指定滤波器的性能指标来快速设计 FIR 或 IIR 滤波器。本节演示如何利用 FDATool 工具来设计一个 FIR 滤波器，并把滤波器系数输出到目标 DSP 中。

① 利用 FDATool 工具来设计滤波器。

在 MATLAB 命令窗中输入 fdatool，打开 FDATool(Filter Design & Analysis Tool)图形设计界面。在此 FDATool 设计界面中指定滤波器设计方法为 FIR(Window)，Window 窗函数选为 Hamming，滤波器类型选为 Lowpass(低通)，滤波器阶数为 16、采样频率 Fs 为 1000 Hz、截止频率 Fc 为 250 Hz，滤波器结构选为直接 I 型。指定完这些滤波器设计参数后，点击 Design Filter，开始生成滤波器系数。图 3.33 为 FDATool 工具的设计界面。

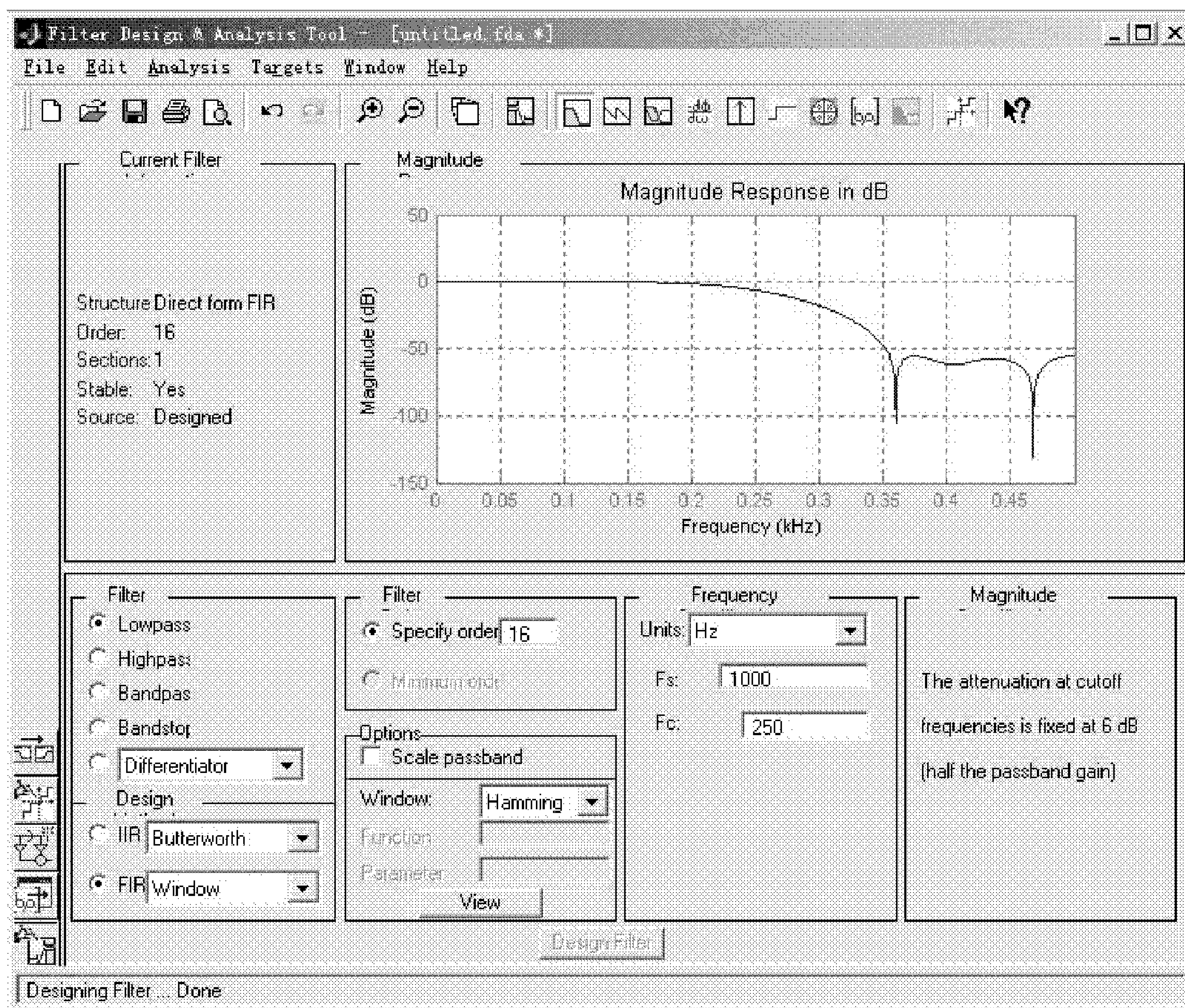


图 3.33　FDATool 工具的设计界面

② 把 FDATool 中设计的滤波器系数输出到一个 C 头文件中。

在 FDATool 设计界面中选择 Targets→Export to Code Composer Studio(tm)IDE，打开 Export to CCS IDE 对话框。在此对话框的 Export mode 项中选择 C header file，C 头文件中的变量名可以选择默认名，输出数据类型选择为 Signed 32 - bit integer，目标选择项中指定板号(DSP Board#)和 DSP 号(DSP Processor#)。指定完这些输出选项后，点击 OK，指定头文

件名 filtercoeff.h 和路径 c:\ti\myproject\volume1\，保存此头文件。生成的 filtercoeff.h 头文件如下所示：

```
/*
* Filter Design and Analysis Tool - Generated Filter Coefficients - C Source
*   Generated by MATLAB - Signal Processing Toolbox
*/
/* General type conversion for MATLAB generated C-code */
#include "tmwtypes.h"
/*
* Expected path to tmwtypes.h
* C:\MATLAB6p52\extern\include\tmwtypes.h
*/
/*
* Warning - Filter coefficients were truncated to fit specified data type.
*    The resulting response may not match generated theoretical response.
*    Use the Filter Design & Analysis Tool to design accurate fixed-point
*    filter coefficients.
*/
  const int BL = 17;
  const int32_T B[17] = {
        0,   - 11231506,          0,    49758852,          0,  - 163152079,
        0,    659629954, 1073741824,   659629954,          0,  - 163152079,
        0,     49758852,          0,  - 11231506,          0
      };
```

③ 把滤波器系数的头文件添加到工程中。

把 MATLAB 目录下的 tmwtypes.h 头文件复制到 c:\ti\myproject\volume1\目录下，因为 filtercoeff.h 头文件中要用到 tmwtypes.h 头文件，否则编译链接时会出错。

在 volume.c 源文件的开始处添加一行语句：

＃include “filtercoeff.h”

重新对工程编译链接后，这些头文件会自动添加到工程中，并会在目标 DSP 中分配相应的存储空间来存储这些滤波器系数。

④ 修改 volume.c 中的 Processing( )子程序，对输入信号进行滤波处理。

利用下面一段 C 程序对输入信号进行滤波处理。

```
static int processing(int *input, int *output)
  {
    int filtertaps[16],y;
    int i, j;
    for(i=0; i<16; i++)
     {
       filtertaps[i]=0;   /*复位滤波器抽头*/
      }
    for(i=0; i<BUFSIZE; i++)
     {
```

```
            filtertaps[0]=input[i];
            for(y=0, j=0; j<BL; j++)
              y=y+filtertaps[j]*B[j];
            output[i]=y;
            for(j=BL- 1; j>=1;   j- - )
                filtertaps[j]=filtertaps[j- 1];        /*偏移滤波器抽头中的信号*/
          }
        return(TRUE);
    }
```

重新编译链接工程，生成可执行代码并把可执行代码加载到目标 DSP 中，再删除所有探点和断点。

利用如下一段 MATLAB 代码向目标 DSP 中输入原始数据，并运行目标 DSP 中的程序，读出 DSP 中的处理结果并与 MATLAB 中的处理结果进行比较。

```
fs=1000;                        %采样频率
f1=100; f2=400; f3=300;         %三个正弦信号频率分别为：100 Hz, 400 Hz, 300 Hz。
x=100*sin(2*pi*f1*[1:100]/fs)+sin(2*pi*f2*[1:100]/fs)+sin(2*pi*f3*[1:100]/fs); %产生输入信号，为
                                                                                  %三个正弦信号之和
cc=ccsdsp('boardnum', 0, 'procnum', 0);    %创建与 CCS IDE 相连接的对象
inp_buffer=createobj(cc, 'inp_buffer');    %创建输入数据变量 inp_buffer 的嵌入式对象
out_buffer=createobj(cc, 'out_buffer');    %创建输出数据变量 out_buffer 的嵌入式对象
B=createobj(cc, 'B');                      %创建滤波器系数变量 B 的嵌入式对象

write(inp_buffer, round(x));               %向目标 DSP 的输入缓冲区写入数据
run(cc);                                   %运行目标 DSP 中的程序
pause(5);                                  %运行一段时间
halt(cc);                                  %停止目标 DSP

figure
inpbuffer=read(inp_buffer);                %读出 DSP 中的输入数据
DSPresult=read(out_buffer);                %读出 DSP 中的输出数据
subplot(2, 2, 1)
plot(inpbuffer);                           %画出 DSP 中的输入信号
title('DSP 输入数据');
subplot(2, 2, 2)
plot(DSPresult);                           %画出 DSP 中的输出滤波结果
title('DSP 滤波结果');

filterCoeff=read(B);                       %把滤波器系数读入到 MATLAB 空间中
c=conv(round(x),filterCoeff);              %利用 MATLAB 来计算滤波
subplot(2, 2, 3)
plot(round(x));                            %画出 MALAB 中的输入信号
title('MATLAB 输入数据');
subplot(2, 2, 4);
plot(c(1:100));                 %画出 MATLAB 中的滤波结果，并与 DSP 中的滤波结果进行比较
title('MATLAB 滤波数据');
```

上述代码首先产生一段滤波器的输入数据，此输入数据为三个正弦波之和，频率分别为 100 Hz、400 Hz、300 Hz，数字信号的采样频率为 1000 Hz。前面在利用 FDATool 设计滤波器系数时，指定的截止频率为 250 Hz，因此 400 Hz 和 300 Hz 的信号分量经过滤波器后都被滤除掉了，最后滤波输出的结果为 100 Hz 的正弦信号。图 3.34 为由这段 MATLAB 代码产生的 figure 图。图 3.34 中将 DSP 的处理结果和 MATLAB 的处理结果画在一起进行对比。从本例可以看出，利用 MATLAB 结合 CCS 来调试目标程序具有很大优点。

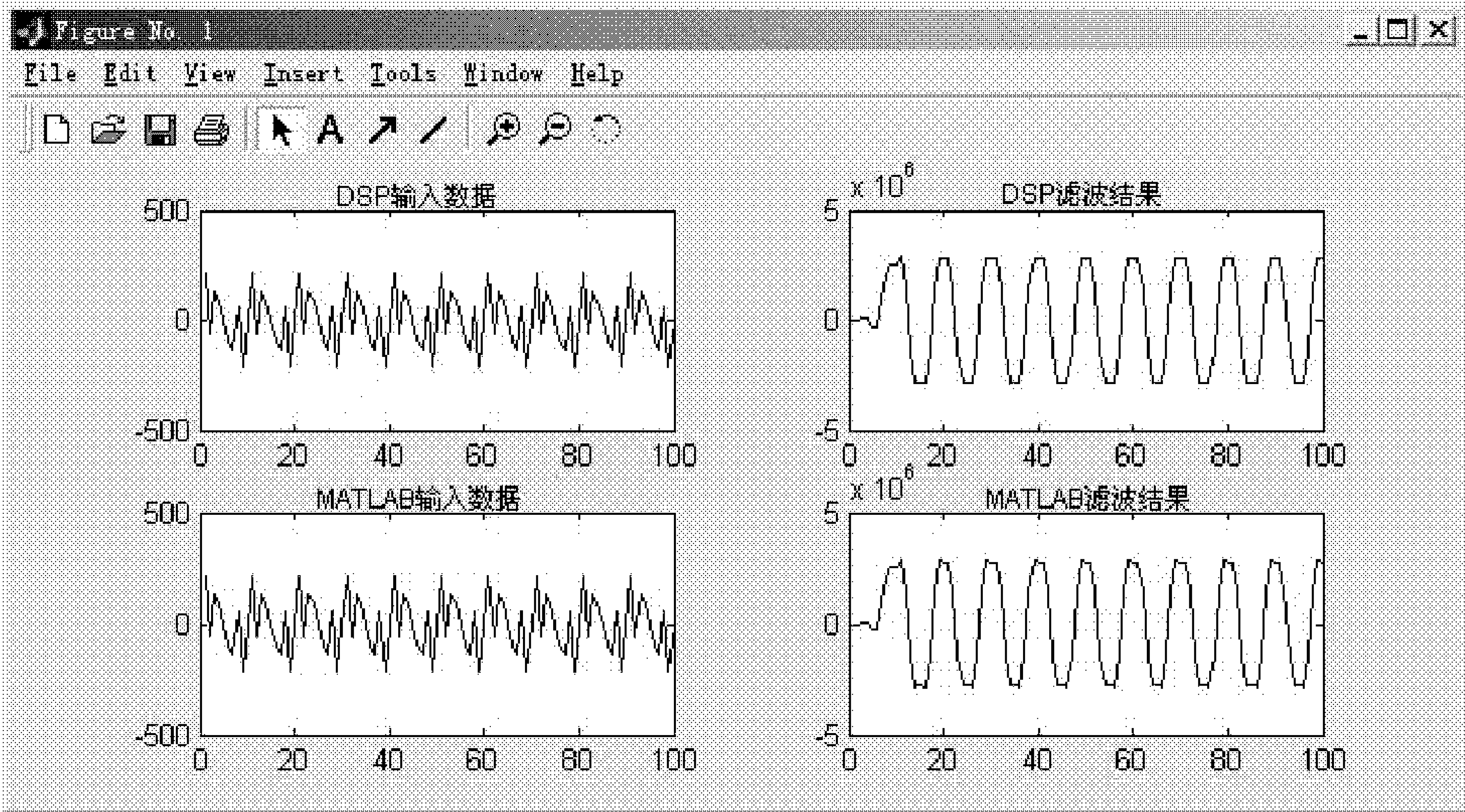


图 3.34　DSP 的处理结果与 MATLAB 的处理结果的比较

3.4　代码实时性分析调试工具

3.4.1　DSP/BIOS 实时操作系统

DSP/BIOS 是一种实时操作系统，它为目标程序提供实时服务，包括实时任务调度、中断、I/O 服务和其它实时操作。利用 DSP/BIOS 的实时分析功能，开发者可以在对代码的实时性能影响最小的情况下，对 DSP 程序的运行进行跟踪、监控、探查，以排除与时间细节有关的问题，而且它也可作为应用程序的一部分，用来操作 I/O 口和外围设备，从而大大简化程序开发过程，缩短产品开发周期。

DSP/BIOS 集成在 CCS 中，在 CCS 下可以方便地向应用程序中添加 DSP/BIOS 功能模块、配置 DSP/BIOS 功能模块及观察实时性分析结果等。在 CCS 中，DSP/BIOS 包含如下三部分：

(1) DSP/BIOS 配置工具。开发者利用此工具来添加和配置应用程序中用到的 DSP/BIOS 功能模块，而且也可用此工具来配置存储器、线程优先级和中断处理程序等。

(2) DSP/BIOS 实时分析工具。利用这些工具可以实时地观察程序的执行情况，从而调试程序的实时性。

(3) DSP/BIOS API 应用程序接口函数。DSP/BIOS 提供了多达 150 个 DSP/BIOS API 函数，这些函数都可以被 C、C++或汇编语言程序调用。这些函数分别包含在 20 多个不同的功能模块(module)中，利用 DSP/BIOS 配置工具把需要的功能模块和目标应用程序链接起来，然后在应用程序中添加调用 API 函数的语句，只有这样这些被调用函数才最终成为可执行代码的一部分(占据非常小的存储器空间和 DSP 运行时间)。

按如下步骤可创建具有 DSP/BIOS 功能模块的可执行代码：

(1) 在 CCS IDE 界面下，选择 File→New→DSP/BIOS Configuration，打开 DSP/BIOS 配置模板窗，如图 3.35 所示。在此模板窗中选择指定的模板。

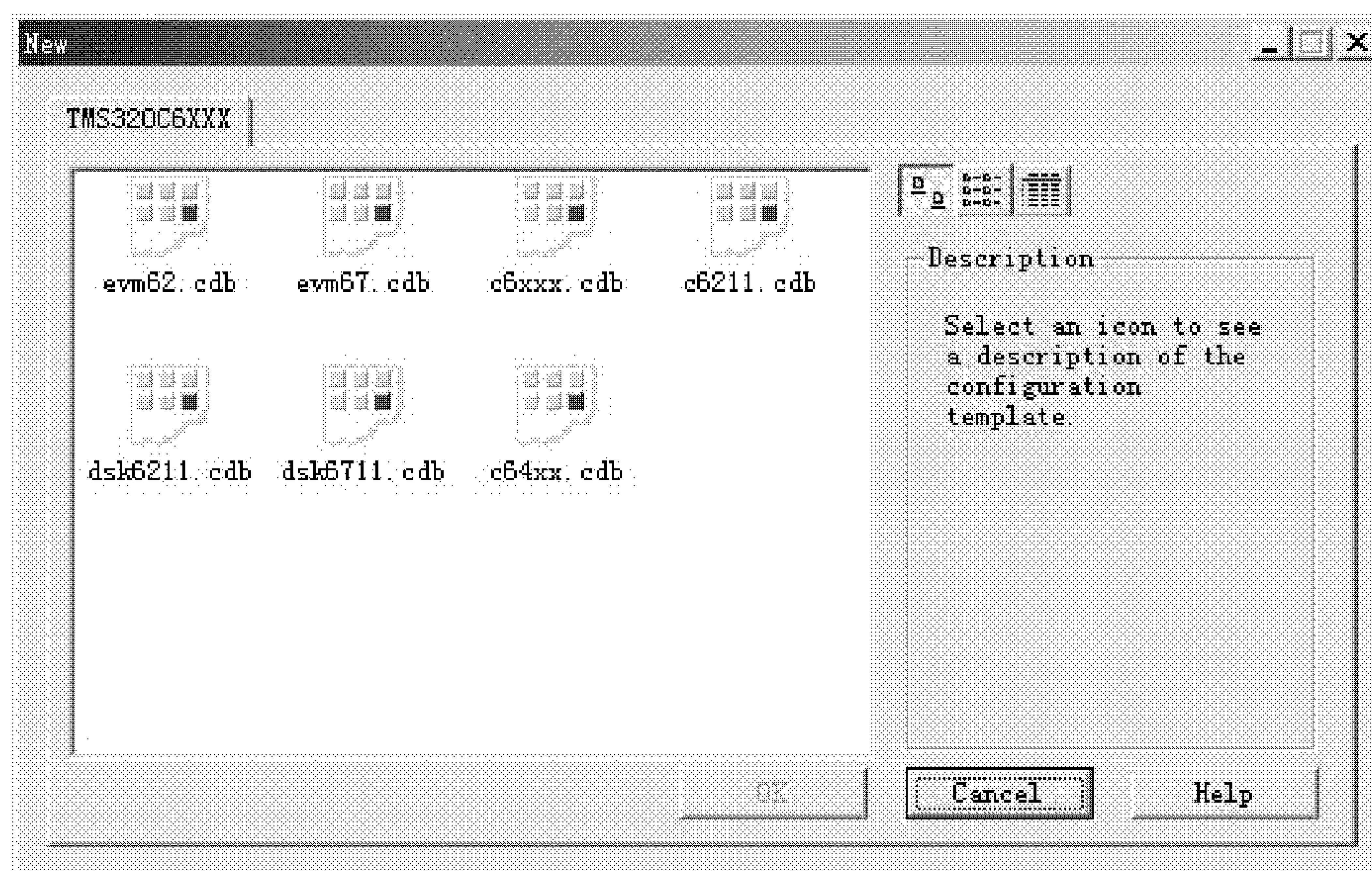


图 3.35　DSP/BIOS 配置模板窗

(2) 在 DSP/BIOS 配置模板窗中选择所需模板，点击 OK 退出 DSP/BIOS 配置模板窗，同时打开一个新的 DSP/BIOS 配置工具窗口，如图 3.36 所示。

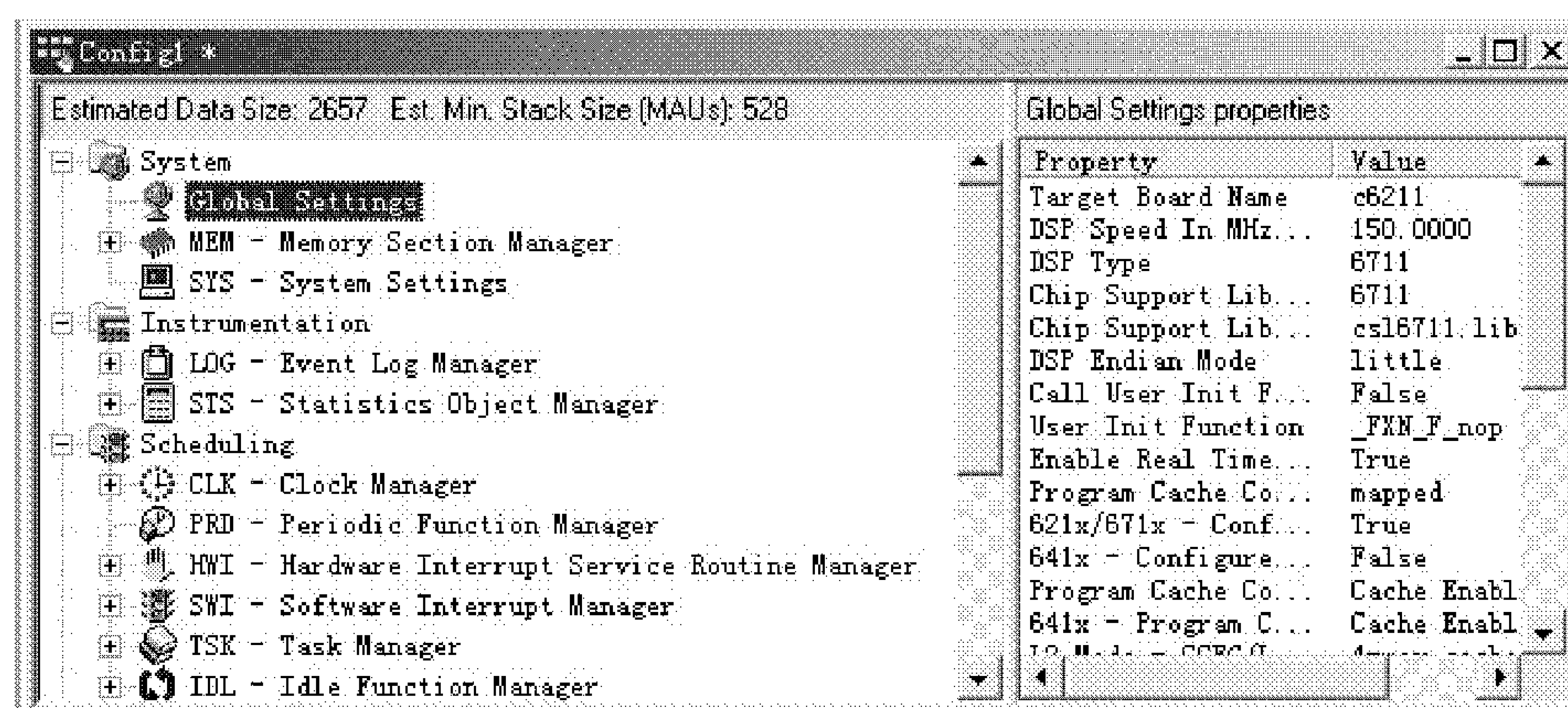


图 3.36　DSP/BIOS 配置工具窗口

DSP/BIOS 配置工具窗口中列出了所有 DSP/BIOS 功能模块，这些功能模块在 DSP/BIOS 配置工具窗口中分别位于不同的文件夹中。

- System——系统操作包括：

Global Setting：设置全局参数，包括：目标板名、CPU 速度、DSP 类型、endian 模式、调用初始化函数、使能实时分析、使能所有的跟踪事件和其它一些 DSP 指定参数。

MEM：指定存储器段，用来定位目标程序中的各种代码和数据段。

SYS：此功能模块提供基本系统服务的通用函数，例如，停止程序的执行和打印文本信息等。

- Instrumentation——事件记录和信息统计包括：

LOG：此事件记录模块能够在目标程序执行过程中实时地捕获事件信息。可以利用系统记录模块(System Logs)，也可以定义用户自己的事件记录模块。利用 CCS IDE 中的 DSP/BIOS 分析工具，可以实时地观察这些记录模块中的信息。

STS：此统计模块控制统计累计器，用来存储目标程序运行过程中的主要统计信息。利用 CCS IDE 中的 DSP/BIOS 分析工具，可以实时地观察这些统计信息。

- Scheduling——任务调度包括：

CLK：此模块控制片内定时器，可以提供具有高分辨率(4 个指令周期)和低分辨率(几个毫秒或更长)的 32 位实时时钟。

PRD：此周期函数模块用来触发目标函数按周期执行。这些函数的执行周期由 CLK 模块的时钟率或通过定期调用 PRD_tick 来控制。

HWI：此模块为硬件中断服务程序提供支持。

SWI：此模块为软件中断提供支持。软件中断具有 15 个优先级，但所有优先级都低于硬件中断的优先级。

TSK：此模块用来控制任务线程，这些任务线程具有比软件中断更低的优先级。

IDL：此模块对 idle 函数进行操作。当目标程序中没有更高优先级的函数要执行时，就会进入 idle 循环。

- Synchronization——同步控制包括：

SEM：此模块对计数旗语(Semaphore)进行操作，计数旗语可以用来进行任务同步和任务互斥。

MBX：此模块用于任务之间的信息传递。

QUE：此模块对数据队列进行操作。

LCK：此模块锁存共享的全局资源，用来仲裁几个竞争的任务对某一资源的访问。

- Input/Output——I/O 服务包括：

RTDX：此模块是实时数据交换模块，用来在主机和目标 DSP 之间实时传递数据。

HST：此主机 I/O 模块用来在主机和目标 DSP 之间传递数据。主机通道被静态配置为输出或输入。

PIP：此模块用来缓存输入、输出数据流。利用这些数据缓冲可以在 DSP 设备和其它外围设备的 I/O 之间提供兼容的软件数据结构。

SIO：此模块提供与设备无关的高效率 I/O 数据流服务。

- CSL - Chip Support Library——芯片支持库。

此芯片支持库中提供了多个C程序函数，用来配置和控制DSP上的外围设备。DSP/BIOS 配置工具提供图形界面来配置和操作芯片的外围设备，从而大大简化实际系统的程序开发

过程、缩短开发时间，而且具有可移植性、标准化和兼容性。CSL 具有以下优点：

利用标准程序对外设进行编程。CSL 为每一片内外设提供了一个高级语言编程接口，包括数据类型和配置外设寄存器的宏，以及对每一外设进行各种操作的函数。

基础资源管理。CSL 通过 open 和 close 函数来对基础资源进行管理。这种操作对支持多通道的外围设备尤其有用。

利用符号来描述外围设备。CSL 产生的代码对所有的寄存器及其位域都用符号进行描述，因此这种高级语言编程方法，可以使目标代码在以后的升级中比较容易。

CSL 中提供了如下模块(不同目标平台所支持的模块有所不同)：CACHE(对数据和程序 cache 进行操作)、CHIP(提供芯片指定的或芯片相关的代码)、CSL(用来初始化库)、DAT(用来产生与 DMA 和 EDMA 设备无关的数据流)、DMA(对 DMA 设备进行操作)、EDMA(对 EDMA 设备进行操作)、EMIF(配置 EMIF 寄存器)、HPI(配置 HPI 寄存器)、IRQ(对 CPU 中断进行操作)、MCBSP(配置 MCBSP 寄存器)、PCI(对 PCI 接口进行操作)、PWR(配置 power－down 逻辑)、TIMER(配置定时器寄存器)、XBUS(配置 XBUS 寄存器)。

(3) 在 DSP/BIOS 配置工具窗中，完成如下任务：

设置应用程序的全局属性：右击 System 目录下的 Global Setting，从弹出的菜单中选择 Properties，在 Global Setting 属性对话框中设置全局属性。

右击目标程序中需要用到的功能块管理器(Manager)，从弹出的菜单中选择 Insert××××(××××为功能块名称)，插入此功能块对象。右击新插入的功能块对象，从弹出的菜单中选择 Rename，可对此功能块对象重新命名(也可以利用默认的对象名，即功能块名+数字)。

最后还要分别设置功能块管理器和功能块对象属性：分别右击功能块管理器和插入的功能块对象，从弹出的菜单中选择 Properties，分别在各自的属性对话框中设置其属性。

(4) 保存此 DSP/BIOS 配置文件。

选择 File→Save as 或点击工具栏中的 Save 图标，指定文件名(比如 *projectname*.cdb，*projectname* 为用户工程名)和保存路径，保存此 DSP/BIOS 配置文件(*.cdb)。保存此配置文件时，同时产生如下文件：

projectname.cdb：此文件保存 DSP/BIOS 的配置设置(*projectname* 为用户在保存配置文件时指定的文件名，下同)。

*projectname*cfg.cmd：链接命令文件。

*projectname*cfg.h：头文件，包含 DSP/BIOS 功能块头文件，并对配置文件中创建的功能块对象的外部变量进行声明。

*projectname*cfg.s62：DSP/BIOS 设置的汇编语言源文件。

*projectname*cfg.h62：*projectname*cfg.s62 中包含的汇编语言头文件。

*projectname*cfg_c.c：定义 CSL 结构和属性的 C 语言源文件。

(5) 向工程中添加 DSP/BIOS 配置文件。

选择 project→Add Files to project，在添加文件窗中选择 *projectname*.cdb(文件类型选择 Configuration File*.cdb)并打开。向工程中添加 DSP/BIOS 配置文件 *projectname*.cdb 后，CCS IDE 会自动向工程中添加如下文件：

projectname.cdb：自动加入到工程视窗的 DSP/BIOS Config 目录下。

*projectname*cfg.s62：自动加入到工程视窗的 Generated Files 目录下。

*projectname*cfg_c.c：自动加入到工程视窗的 Generated Files 目录下。

继续向工程中添加其它文件，在添加文件窗中选择 *projectname*cfg.cmd(文件类型选择 Linker Command File*.cmd)并打开。如果工程中已经包含一个链接命令文件(*.cmd)，则当向此工程中添加另一个链接命令文件时，CCS IDE 会弹出一个警告对话框，此警告对话框询问用户是否替代以前的链接命令文件，因为每一工程中只能包含一个链接命令文件。点击 Yes，替换以前的链接命令文件并退出此警告对话框。

(6) 删除工程中的部分文件。

如果工程中包含 vectors.asm 文件，此文件中定义了中断矢量表，则当工程中添加 DSP/BIOS 功能时，这些中断矢量表会自动转移到 DSP/BIOS 配置文件中，因此工程中不再需要此文件。右击工程视窗中的 vectors.asm 文件，从弹出的菜单中选择 Remove from project，删除此文件。

如果工程中包含 rts××××.lib 文件(××××为用户的目标 DSP)，此库文件也被自动包括在链接命令文件 *projectname*cfg.cmd 中，工程不再需要此文件。右击工程视窗中的 rts××××.lib 文件，从弹出的菜单中选择 Remove from project，删除此文件。

(7) 修改源代码，调用 DSP/BIOS API 函数。

双击工程视窗中的主函数源文件，在 CCS IDE 的源代码编辑窗中打开此源文件。按如下格式在源代码中添加 DSP/BIOS API 函数调用：

```
/* DSP/BIOS 头文件*/
#include<std.h>      /*所有调用 DSP/BIOS API 的程序都必须包含此头文件，而且必须位于
                      其它 DSP/BIOS 功能块头文件的前面*/
#include<log.h>      /*DSP/BIOS 的 LOG 模块头文件*/
#include<swi.h>      /*DSP/BIOS 的 SWI 模块头文件*/
   ⋮                        ⋮
/*以下为应用程序的主体*/
   ⋮
LOG_printf(&trace, "hello world");   /*调用 LOG 模块中的 LOG_printf 函数来打印一字符串
                                       信息，trace 为在 DSP/BIOS 配置文件中定义的 LOG
                                       功能块对象*/
   ⋮
SWI_dec(&processing_SWI);   /*调用 SWI 模块中的 SWI_dec 函数，processing_SWI 为在
                              DSP/BIOS 配置文件中定义的 SWI 功能块对象*/
   ⋮
```

说明：对用户程序中用到的每一个 DSP/BIOS 功能模块，在源文件的开头都必须对其头文件进行包含(#include)，而且对于 C 语言程序，必须首先包含 std.h 头文件，然后再包含其它的 DSP/BIOS 功能模块头文件×××.h(×××为功能模块名)。std.h 头文件中定义了标准类型和常数。对于汇编语言程序，包含的 DSP/BIOS 功能模块头文件为×××.h62(×××为 DSP/BIOS 功能模块名，尽管头文件后缀为 h62，但同样可应用于 C67××DSP)。

对于每一个 DSP/BIOS 功能模块中的所有 API 函数，用户可以查看 CCS IDE 的在线帮助，本书不再对其一一介绍。在 3.4.3 节的例子中本书对部分常用的 DSP/BIOS API 函数进行了演示。

(8) 保存修改后的源代码，重新编译链接工程并加载运行。

(9) 打开 DSP/BIOS 实时分析工具来实时观察目标程序的运行情况。

在 CCS IDE 界面下，选择 DSP/BIOS→DSP/BIOS 实时分析工具，或点击 DSP/BIOS 实时分析工具栏中的相应图标。CCS IDE 提供的 DSP/BIOS 实时分析工具如下：

CPU Load Graph：显示目标 CPU 的负荷曲线图。

Execution Graph：显示目标程序中各线程的执行情况(根据 LOG_system 系统对象中记录的信息进行显示)。

Host Channel Control：把主机文件与 HST 对象连接起来，并开始数据传递。

Message Log：显示 LOG 对象中记录的文本信息。

Statistics View：显示 STS、PIP、PRD、SWI、TSK 和 HWI 对象的统计信息。

RTA Control Panel：利用此面板使能或禁止对各种 DSP/BIOS 功能块的记录和统计。

Kernel/Object View：观察目标程序中当前 DSP/BIOS 对象的配置、状态等信息。

需要说明一点，这里指的实时观察确切地说应该是准实时的，DSP/BIOS 模块对目标程序中信息的获取是实时的，但信息向主机的传送(利用 RTDX 技术，下节再介绍)并非严格意义上的实时。

(10) 根据 DSP/BIOS 的实时分析结果重新修改目标程序，重新编译链接并加载运行。

3.4.2　RTDX 实时数据交换

利用 RTDX 技术，可以在不停止目标 DSP 中程序执行的前提下，完成主机与目标 DSP 之间的实时数据交换。DSP/BIOS 的实时调试、分析工具就是利用 RTDX 技术在主机上实时地获取和监视目标 DSP 中程序执行信息的。用户也可以利用 RTDX 技术向目标 DSP 实时地发送数据、获取处理结果并进行实时分析和可视化。

RTDX 实时数据交换技术是利用目标 DSP 的片上仿真逻辑和 JTAG 接口实现的，它不占用 DSP 的系统总线、串口等 I/O 资源，所以数据传递可以在不中断目标程序执行的前提下实时进行，对目标程序的影响很小。

RTDX 包括主机和目标 DSP 两部分，如图 3.37 所示。

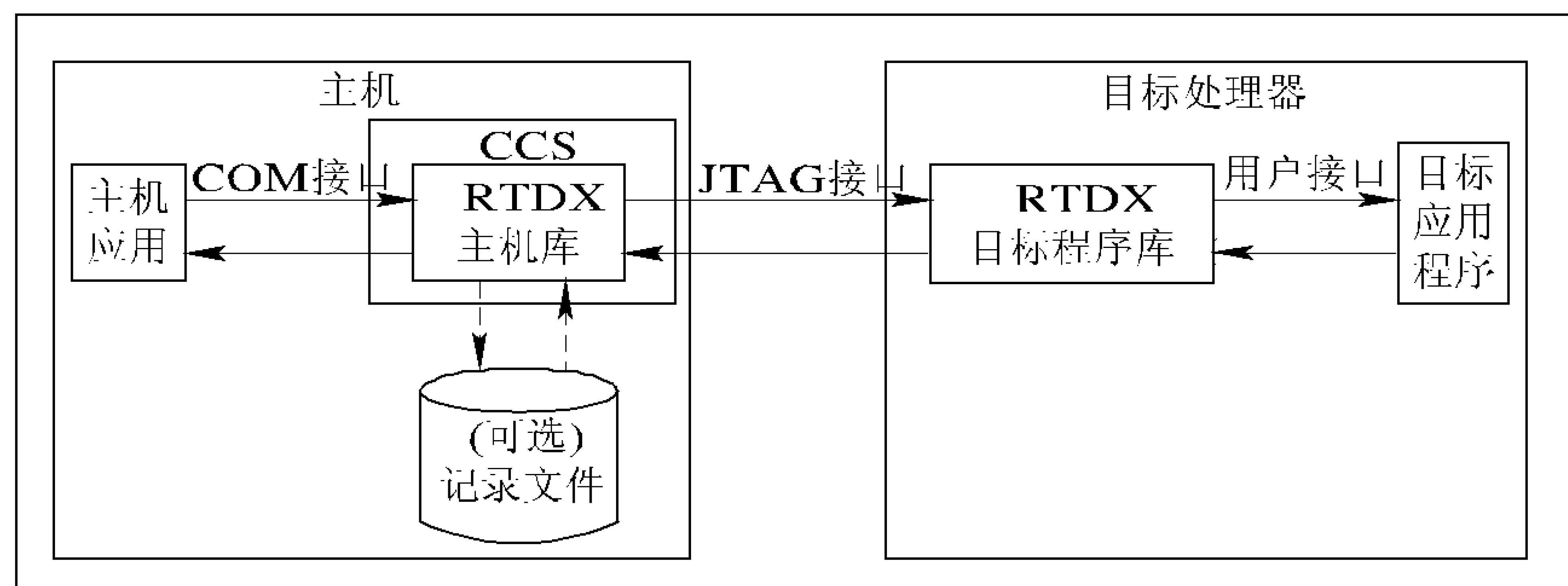


图 3.37　RTDX 的组成

一个很小的 RTDX 目标程序库运行在目标 DSP 中，目标应用程序调用 RTDX 库中的 API 函数，向其发送数据或接收其数据。RTDX 目标库函数利用硬件仿真器(Emulator)经过 JTAG 接口向主机平台发送或接收数据。

在主机平台上，RTDX主机库与CCS IDE配合操作。数据可视化和分析工具通过COM接口与RTDX主机库进行通信，向其发送或接收数据。RTDX主机库支持两种从目标DSP接收数据的模式：非连续模式和连续模式。在非连续模式下，从目标DSP接收到的数据被写入到主机的一个文件中，此模式仅当开发者只需获取有限数据时才使用。在连续模式下，从目标DSP接收到的数据只是简单地被RTDX主机库缓存，并不写入到主机文件中，如果开发者需要连续地得到和可视化从目标DSP中接收到的数据并且不需要把此数据保存在主机文件中，就使用此模式。

CCS IDE中集成了三种RTDX工具，用来图形化配置RTDX、设置RTDX通道和对RTDX工作情况进行诊断。利用这些工具可以帮助开发者方便地使用RTDX功能。CCS IDE提供的RTDX工具如下：

• RTDX诊断工具(Diagnostics Control)

选择Tools→RTDX→Diagnostics Control，打开RTDX诊断窗口。此RTDX诊断工具测试目标DSP到主机和主机到目标DSP的数据传递情况，从而确定RTDX是否工作正常。

• RTDX配置工具(Configuration Control)

选择Tools→RTDX→Configuration Control，打开RTDX配置窗口。此RTDX配置工具利用图形界面来完成配置和控制RTDX，利用此工具可以完成如下任务：观察RTDX当前配置；使能或禁止RTDX；在RTDX的配置属性对话框中重新配置RTDX和选择端口设置。

• RTDX通道观察工具(Channel Viewer Control)

选择Tools→RTDX→Channel Viewer Control，打开RTDX通道观察窗口。此RTDX通道观察工具用来设置RTDX通道，利用此工具可以完成以下任务：向RTDX通道观察列表中添加或删除某一RTDX通道；使能或禁止列表中的RTDX通道。

按以下步骤可实现主机与目标DSP之间的实时数据交换RTDX：

(1) 创建RTDX通道。

利用DSP/BIOS的配置工具(如果工程中没有DSP/BIOS配置文件，选择File→New→DSP/BIOS Configuration，打开一个新的配置文件；如果工程中已有DSP/BIOS配置文件，在CCS IDE中打开它)插入RTDX通道对象，或在源代码文件中加入如下创建RTDX通道的语句：

```
RTDX_CreateInputChannel(ichannel);      /*创建一个通道名为ichannel的RTDX输入通道*/
RTDX_CreateOutputChannel(ochannel);  /*创建一个通道名为ochannel的RTDX输出通道*/
```

(2) 使能RTDX通道。

开发者既可以在目标程序中使能RTDX通道，也可以在主机端使能RTDX通道。在目标程序中通过如下语句来使能RTDX通道：

```
RTDX_enableInput(&ichannel);         /*使能RTDX输入通道ichannel*/
RTDX_enableOutput(&ochannel);      /*使能RTDX输出通道ochannel*/
```

在主机端使能RTDX通道的方法很多，本书推荐读者利用MATLAB对RTDX通道进行操作，下面的MATLAB语句使能RTDX通道：

```
cc.rtdx.open('ichannel', 'w');          % cc为创建的连接对象，打开RTDX输入通道ichannel
cc.rtdx.enable('ichanne');               %使能RTDX通道ichannel
cc.rtdx.enable;                          %使能RTDX接口
```

(3) 在利用 RTDX 通道传送数据前，必须确定此通道已使能。

在目标程序中利用下面的函数判断 RTDX 通道是否已使能：

```
RTDX_isInputEnabled(&ichannel)
RTDX_isOutputEnabled(&ochannel)
```

如果通道已使能，上面的函数返回 TRUE，否则返回 FALSE。

主机向 RTDX 通道发送或从 RTDX 通道接收数据前，也要首先确定此通道已使能并且可读写，利用下面的一段 MATLAB 程序进行判断：

```
cc.rtdx.isenabled('ichannel');     %检查 ichannel 通道是否已使能，是返回 1，否返回 0
cc.rtdx.iswritable('ichannel');    %检查 ichannel 通道是否可写，是返回 1，否返回 0
```

(4) 利用定时查询法从 RTDX 通道中读入数据或向 RTDX 通道中写入数据。

在目标程序中加入下面的一段程序来从 RTDX 通道中读入数据，经过处理之后再把处理结果写入到 RTDX 输出通道中。

```
if(RTDX_isInputEnabled(&ichannel))
{
 RTDX_read(&ichannel, input, size*sizeof(datatype));  /*从 RTDX 输入通道中读入数据，ichannel
        为通道名，input 为输入数据的首地址，size 为输入数据的长度，datatype 为数据类型*/
}
Data_Processing(input, output);     /*用户编写的数据处理程序，对输入数据进行处理*/
if(RTDX_isOutputEnabled(&ochannel))
{
 RTDX_write(&ochannel, output, size*sizeof(datatype));  /*ochannel 为 RTDX 输出通道名，output
        为输出数据的首地址，size 为输出数据长度，datatype 为数据类型*/
 while(RTDX_writing) { RTDX_poll( ); }            /*等待数据输出*/
 }
```

主机向 RTDX 通道发送或从 RTDX 通道接收数据，下面的一段 MATLAB 程序实现向 RTDX 通道写入数据或从 RTDX 通道读出数据的功能：

```
input_data=cc.rtdx.readmsg('ochannel','datatype',1);   %从 ochannel 通道中读入一个信息，并放在
      MATLAB 空间的 input_data 变量中
cc.rtdx.writemsg('ichannel', input_data);        %向 ichannel 通道中写入数据 input_data，向 RTDX
      输入通道写入数据前，input_data 必须经过数据类型转换
```

(5) 保存工程中修改过的所有文件，重新编译链接并加载运行。

(6) 利用主机程序向 RTDX 通道写入数据，或从 RTDX 通道中读出数据再进行分析处理。

3.4.3　应用 DSP/BIOS 调试代码实时性演示例子

在 3.3.2 节的演示例子中，我们使用了 CCS 演示目录中的 volume.pjt 工程例子，在本节我们仍以此例为基础，添加 DSP/BIOS 功能模块，演示 DSP/BIOS 的使用过程。读者也可以直接把 c:\ti\tutorial\××××\volume2\目录下的所有内容复制到用户目录 c:\ti\myproject\volume2\下。本节仍以前面已经演示过的 volume1 目录中的工程(非 DSP/BIOS 工程)为例，通过修改其源代码和添加 DSP/BIOS 功能块对象后，使其变成 DSP/BIOS 工程，从而可以对代码的实

时性进行调试，以排除程序中的时间瓶颈问题。

在 c:\ti\myproject\目录下创建一个新文件夹 volume2，把 c:\ti\tutorial\××××\volume1\目录下的所有内容复制到新创建的目录 c:\ti\myproject\volume2\下。

按以下步骤完成演示：

(1) 打开 CCS IDE，选择 Project→Open，把 c:\ti\myproject\volume2\目录下的 volume.pjt 工程加载到 CCS IDE 中。

(2) 选择 File→New→DSP/BIOS Configuration，指定 DSP/BIOS 配置模板后打开 DSP/BIOS 配置工具窗口。

打开 DSP/BIOS 配置工具窗口中的 Global Settings 属性对话框，在此属性对话框中指定目标板的名称、目标 DSP 的指令时钟频率、目标 DSP 的类型等，并且选择 Enable Real Time Analysis 和 Enable All TRC Trace Event Class。

右击 LOG－Event Log Manager，从弹出的菜单中选择 Insert LOG，添加一个新的 LOG 对象。右击此新添加的 LOG 对象，从弹出的菜单中选择 Rename，给此 LOG 对象改名为 trace。

右击 LOG_system 对象，从弹出的菜单中选择 Properties，在 LOG_system 属性对话框的 buflen(words)项中选择 512，然后点击 OK 退出此属性对话框。

右击 CLK－Clock Manager，从弹出的菜单中选择 Insert CLK，添加一个新的 CLK 对象。右击此新添加的 CLK 对象，从弹出的菜单中选择 Rename，给此 CLK 对象改名为 dataIO_CLK。修改 dataIO_CLK 对象的属性，右击 dataIO_CLK，从弹出的菜单中选择 Properties，在 dataIO_CLK 属性对话框的 function 项中输入：_dataIO。dataIO 为用户编写的 C 语言函数，在 volume.c 源代码中可以看到此函数，dataIO 函数前的下划线表明此函数为 C 语言函数，因为 DSP/BIOS 配置以汇编语言保存，而在汇编代码中调用的 C 函数前必须加一下划线。dataIO_CLK 每隔 1 ms(默认为 1 ms，用户也可以在 CLK－Clock Manager 属性对话框中设置为其它时间间隔)执行一次 dataIO 函数(在 CLK－Clock Manager 属性对话框中可以看到 CLK 功能块依靠 HWI_INT14 中断来调用函数执行，而 HWI_INT14 的中断源分配给 Timer 0。当 Timer 0 产生中断时调用 CLK_F_isr 函数，在 CLK_F_isr 函数的执行过程中再调用 CLK 对象中的用户函数)。

右击 SWI－Software Interrupt Manager，从弹出的菜单中选择 Insert SWI，添加一个新的 SWI 对象。右击此新添加的 SWI 对象，从弹出的菜单中选择 Rename，给此 SWI 对象改名为 processing_SWI。修改 processing_SWI 对象的属性，右击 processing_SWI，从弹出的菜单中选择 properties，在 processing_SWI 属性对话框中设置参数，如图 3.38 所示。



图 3.38　processing_SWI 的属性对话框

• function：当此软件中断发生时调用此项中指定的函数，本例中的 processing 为用户编写的 C 语言函数，在下面的 volume.c 代码中可以看到此函数。

• mailbox：此值用来控制软件中断的发生频率，本例中设置为 10。volume.c 中的 dataIO

函数调用 DSP/BIOS API 函数 SWI_dec，SWI_dec 函数每调用一次都会将 mailbox 中的值减 1，当 mailbox 中的值减到 0 时就会产生 SWI 中断，从而执行 processing 函数，因此 processing 函数的执行间隔时间为 10 ms(因为 dataIO 的默认调用时间间隔为 1 ms)。

• arg0, arg1：function 中函数的输入参数。本例中的 inp_buffer 和 out_buffer 都是 volume.c 中的全局变量，用来作为 processing 函数的输入参数。

(3) 选择 File→Save，指定文件名为 volume.cdb，目录为 c:\ti\myproject\volume2\，保存此 DSP/BIOS 配置文件，CCS 自动在此目录下保存如下文件：volume.cdb、volumecfg.cmd、volumecfg.h、volumecfg.s62、volumecfg.h62、volumecfg_c.c。

(4) 向工程中添加 DSP/BIOS 文件。

选择 Project→Add Files to Project，在打开的添加文件窗中选择 volume.cdb(文件类型选择 Configuration File*.cdb)并打开。CCS IDE 会自动向工程中添加如下文件：volume.cdb、volumecfg.s62、volumecfg_c.c。

继续向工程中添加 volumecfg.cmd(文件类型选择 Linker Command File*.cmd)文件，在弹出的警告对话框中点击 Yes，替换以前的链接命令文件 volume.cmd。

(5) 删除工程中不再需要的文件。

右击工程视窗中的 vectors.asm 文件，从弹出的菜单中选择 Remove from project，删除此文件。利用同样的方法删除 rts××××.lib 文件(××××为用户的目标 DSP)。

(6) 修改源代码。

双击工程视窗中的 volume.c 文件，在 CCS IDE 的源代码编辑窗中打开此源文件，修改其源代码。用户也可以直接把 c:\ti\tutorial\××××\volume2\目录下的 volume.c 文件复制到 c:\ti\myproject\volume2\下，替换此目录中以前的 volume.c 文件。经过修改后的 volume.c 中的源代码如下所示。

```
#include <std.h>
#include <log.h>
#include <swi.h>
#include "volumecfg.h"
#include "volume.h"

/* Global declarations */
Int inp_buffer[BUFSIZE];              /* processing data buffers */
Int out_buffer[BUFSIZE];
Int gain = MINGAIN;                   /* volume control variable */
Uns processingLoad = BASELOAD;        /* processing routine load value */

  /* Functions */
extern Void load(Uns loadValue);
Int processing(Int *input, Int *output);
Void dataIO(Void);
Void loadchange(Void);

    /* ======== main ========*/
Void main( )
```

```
{
    LOG_printf(&trace,"volume example started\n");
    /* fall into DSP/BIOS idle loop */
    return;
  }

  /*======== processing ========*/
Int processing(Int *input, Int *output)
  {
      Int size = BUFSIZE;
      while(size- - ){
           *output++ = *input++ * gain;
          }
       /* additional processing load */
       load(processingLoad);
       return(TRUE);
     }
  /*======== dataIO ========*/
Void dataIO()
  {
    SWI_dec(&processing_SWI);    /* post processing_SWI software interrupt */
  }
```

说明：主函数 main 只是调用一次 DSP/BIOS API 函数 LOG_printf( )，用来输出一段字符串信息，然后就进入 DSP/BIOS 的 idle 状态中，等待任何中断的发生。processing( )子函数用来对输入数据乘上某一增益后把结果放到输出地址上，之后再调用 load( )汇编语言函数用来消耗 CPU 时间，load( )函数只是用来消耗 CPU 时间，并不作任何处理。processing( )函数是靠 processing_SWI 软件中断来调用的。dataIO( )子函数只是用来调用 DSP/BIOS API 函数 SWI_dec 的，SWI_dec 将 processing_SWI 属性中的 mailbox 值减 1，当 mailbox 值减到 0 后会触发软件中断 processing_SWI，而 dataIO( )函数的执行是靠 CLK 的对象 dataIO_CLK 来周期触发的(默认为 1 ms)。因此 dataIO( )函数每隔 1 ms 执行一次，而 processing( )函数每隔 10 ms(mailbox 值为 10)执行一次。

(7) 保存修改后的 volume.c 文件，重新编译链接工程并把生成的可执行代码加载到目标 DSP 中。

(8) 选择 Profiler→Start New Session，开始新的性能统计任务。输入任务名后点击 OK，打开统计观察窗口。把 volume.c 中的下列语句行用鼠标直接拖到统计观察窗口的 Ranges 中。

```
LOG_printf( &trace, "volume example started\n");
```

点击 DSP/BIOS 工具栏中的 Open Message Log 图标，打开 Message Log 显示窗口，从 Message Log 窗的 Name 列表中选择 trace 进行显示。

(9) 运行加载在目标 DSP 中的可执行代码，观察 profiler 窗中的统计结果，如图 3.39 所示。与前面对 puts( )函数的统计结果比较，可以看出 LOG_printf 函数的执行时间远远小于 puts( )函数，从而说明了 DSP/BIOS API 函数只占用很少的 CPU 时间，对应用程序的实时性影响很小。

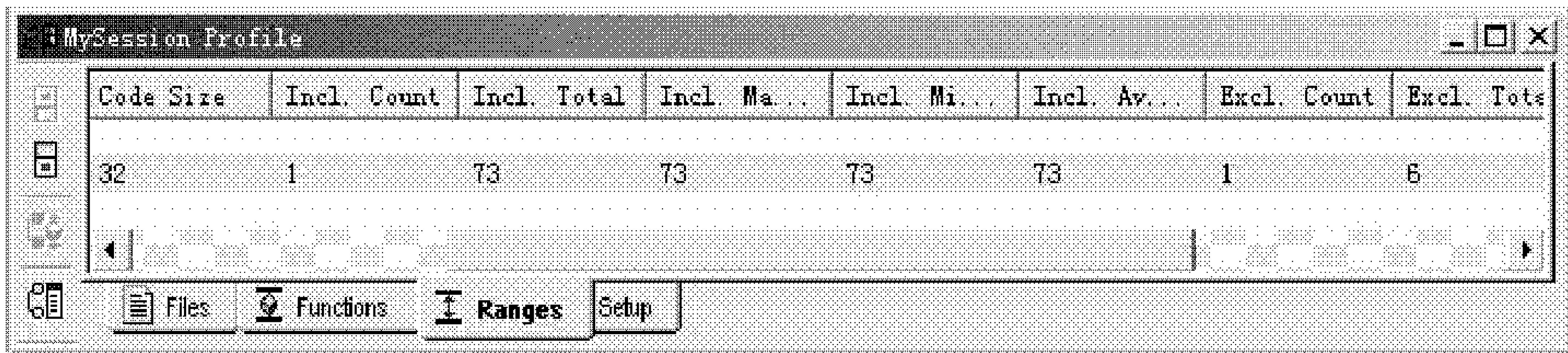


图 3.39　LOG_printf 函数的统计结果

在 Message Log 窗口中显示 volume example started 字符串信息，如图 3.40 所示。



图 3.40　Message Log 窗口的显示结果

(10) 点击 DSP/BIOS 工具栏中的 Open RTA Control Panel 图标，打开 RTA Control Panel 面板，在 RTA Control Panel 面板中选择 enable SWI logging、enable CLK logging、enable SWI accumulators 和 global host enable。右击 RTA Control Panel 面板，从弹出的菜单中选择 Property Page，修改 Message Log/Execution Graph 的刷新率为 1 s。

(11) 统计目标程序中各线程的指令周期数。

点击 DSP/BIOS 分析工具栏中的 Open Statistics View 图标，打开 Statistics View 窗口，用来显示目标程序中各线程的统计信息，右击 Statistics View 窗，从弹出的菜单中选择 Property Page，打开 Statistics View 属性对话框，在此对话框中高亮选中 processing_SWI 对象，点击 OK 退出此对话框。在 Statistics View 窗中显示 processing_SWI 对象的统计结果，如图 3.41 所示。

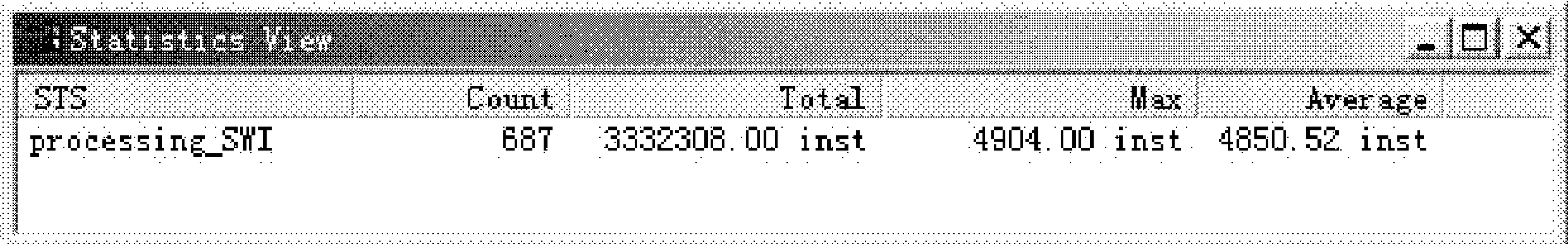


图 3.41　Statistics View 窗中的统计结果

(12) 点击 DSP/BIOS 工具栏中的 Open Execution Graph 图标，打开 Execution Graph 窗口，此窗口显示目标程序中各线程的执行时序，如图 3.42 所示。

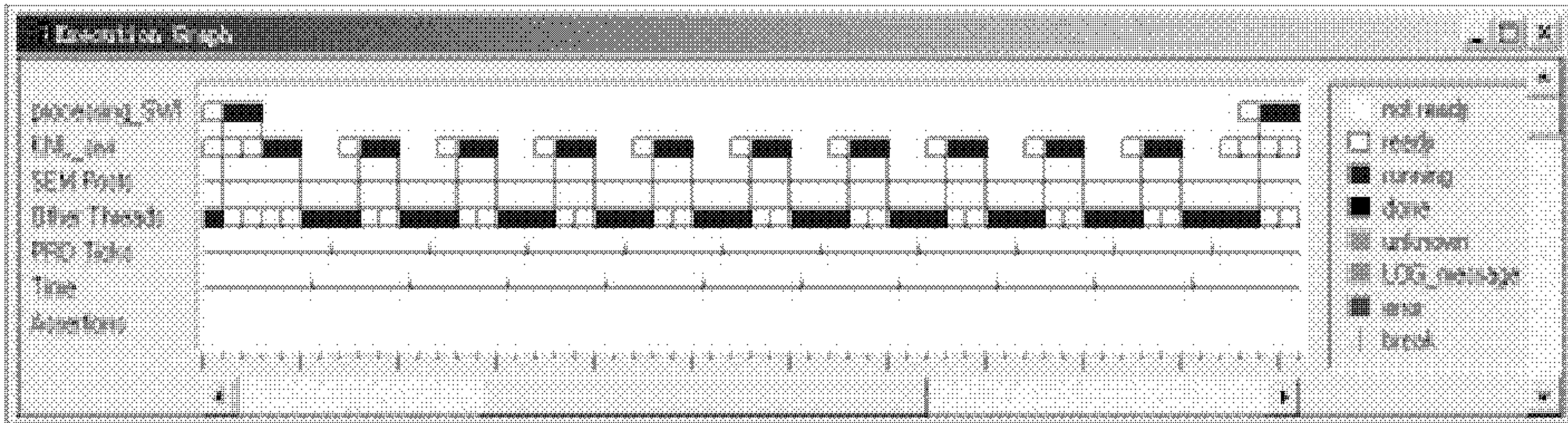


图 3.42　Execution Graph 显示的各线程执行时序(processingLoad 值为 1)

观察图 3.42 可看到，Time 轴上的短竖线表示 CLK 对象的运行情况，两次 processing_SWI 对象的运行时间间隔中有 10 个短竖线，这表明 dataIO_CLK 对象每运行 10 次，processing_SWI 对象运行 1 次，根据前面的分析可知这正是所期望的，即目标程序中不存在与时间有关的问题。

(13) 点击 DSP/BIOS 工具栏中的 Open CPU Load Graph 图标，打开 CPU Load Graph 窗口，此窗口中显示目标 CPU 的负荷曲线。

(14) 利用 Watch Window 窗口修改变量 processingLoad 的值，从而可以改变 processing( )函数的执行时间。

在 volume.c 的源代码编辑窗中高亮选中 processingLoad 变量，右击鼠标，从弹出的菜单中选择 Add To Watch，把 processingLoad 变量添加到 Watch Window 窗中。在 Watch Window 窗中就可以修改变量 processingLoad 的值。修改变量 processingLoad 的值为 200，右击 Execution Graph 窗，从弹出的菜单中选择 Clear，清除先前的显示结果。processingLoad 的值为 200 时，Execution Graph 窗的显示结果如图 3.43 所示。

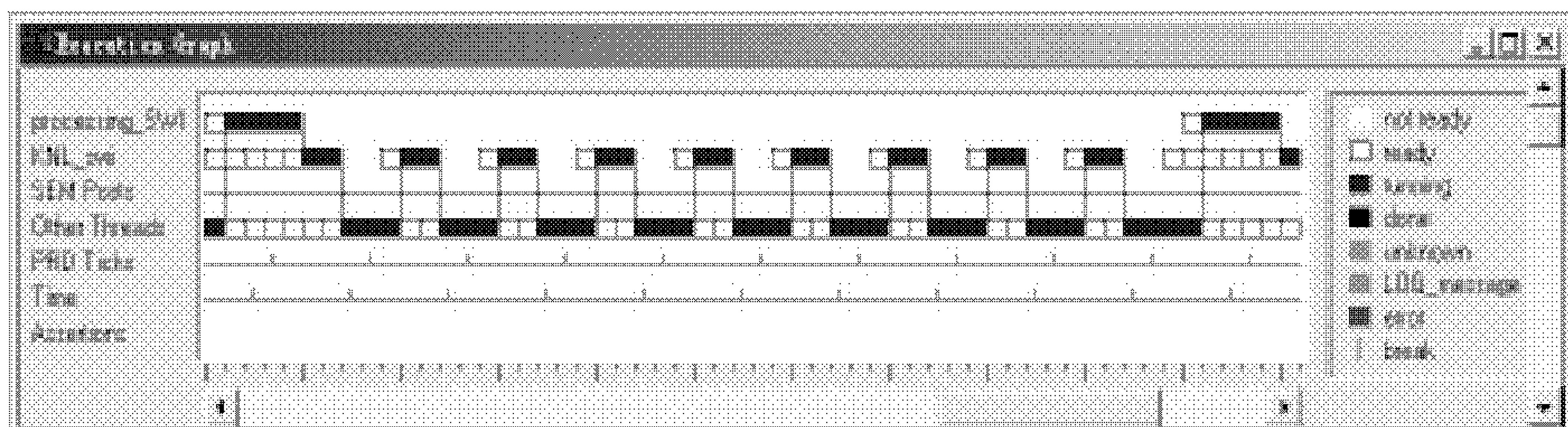


图 3.43 Execution Graph 显示的各线程执行时序(processingLoad 值为 200)

从图 3.43 可以看出，processing_SWI 对象占用更多的执行时间，而且在 processing_SWI 对象的执行过程中发生过 1 至 2 次 dataIO_CLK 对象的执行(这是因为 dataIO_CLK 是靠硬件中断来运行的，因此 dataIO_CLK 对象能够中断 processing_SWI 对象的执行，当 dataIO_CLK 对象的执行完成后再继续处理 processing_SWI 对象)，但目标程序中的各线程仍然满足时间要求。

继续改变 processingLoad 的值为 1250，此时的 Execution Graph 窗和 CPU Load Graph 窗的显示结果分别如图 3.44 和图 3.45 所示。当 processingLoad 的值为 1250 时，CPU 负荷达到了 95%，但从图 3.44 可以看出目标程序的各线程仍然满足时间要求，因为 processing_SWI 对象的执行仍然可以在 10 个 CLK 对象的执行时间内完成。

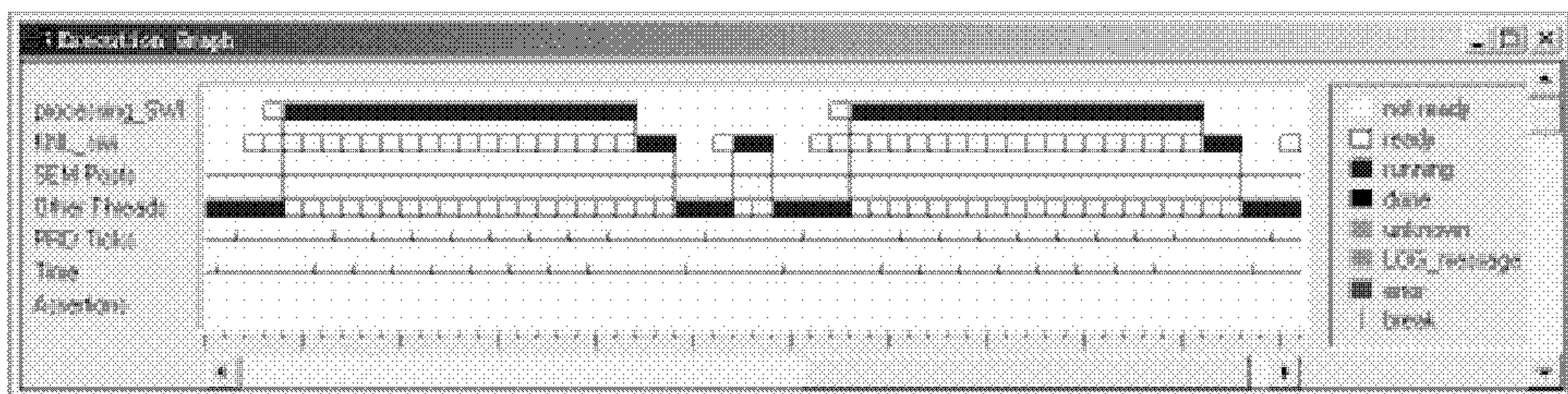


图 3.44 Execution Graph 显示的各线程执行时序(processingLoad 值为 1250)

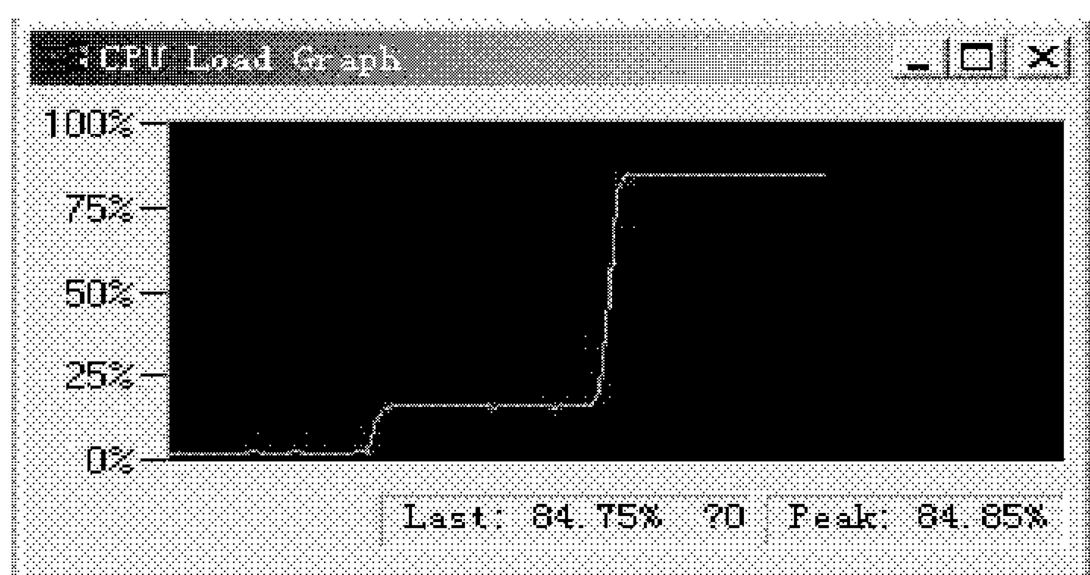


图 3.45 CPU Load Graph 窗的显示结果

继续改变 processingLoad 的值为 1500，此时 Execution Graph 窗和 CPU Load Graph 窗已停止刷新，这是因为向主机发送 Execution Graph 和 CPU Load Graph 更新数据的任务具有最低的优先级，而此时的较高优先级线程已占据了所有目标 CPU 时间，因此向主机发送更新数据的任务无法再执行，此时目标程序中的线程不再满足时间要求。

(15) 在前面我们利用 Watch Window 窗来修改 processingLoad 变量的值，从而改变 processing_SWI 对象的执行时间，然而这会暂时中断目标 DSP 中程序的执行，对目标程序的实时性产生影响。而在接下来的演示过程中，我们修改源代码文件和 DSP/BIOS 配置文件，利用 RTDX 技术来实时修改变量 processingLoad 的值，从而使目标 DSP 中正在执行的程序不会受到影响。

(16) 修改 DSP/BIOS 配置文件。

在工程视窗中，双击 volume.cdb 文件，打开此文件。右击 PRD－Periodic Function Manager，从弹出的菜单中选择 Insert PRD，添加一个新的 PRD 对象。右击此新添加的 PRD 对象，从弹出的菜单中选择 Rename，给此新 PRD 对象改名为 loadchange_PRD，再右击 loadchange_PRD 对象，从弹出的菜单中选择 Properties，修改此对象的属性，如图 3.46 所示。

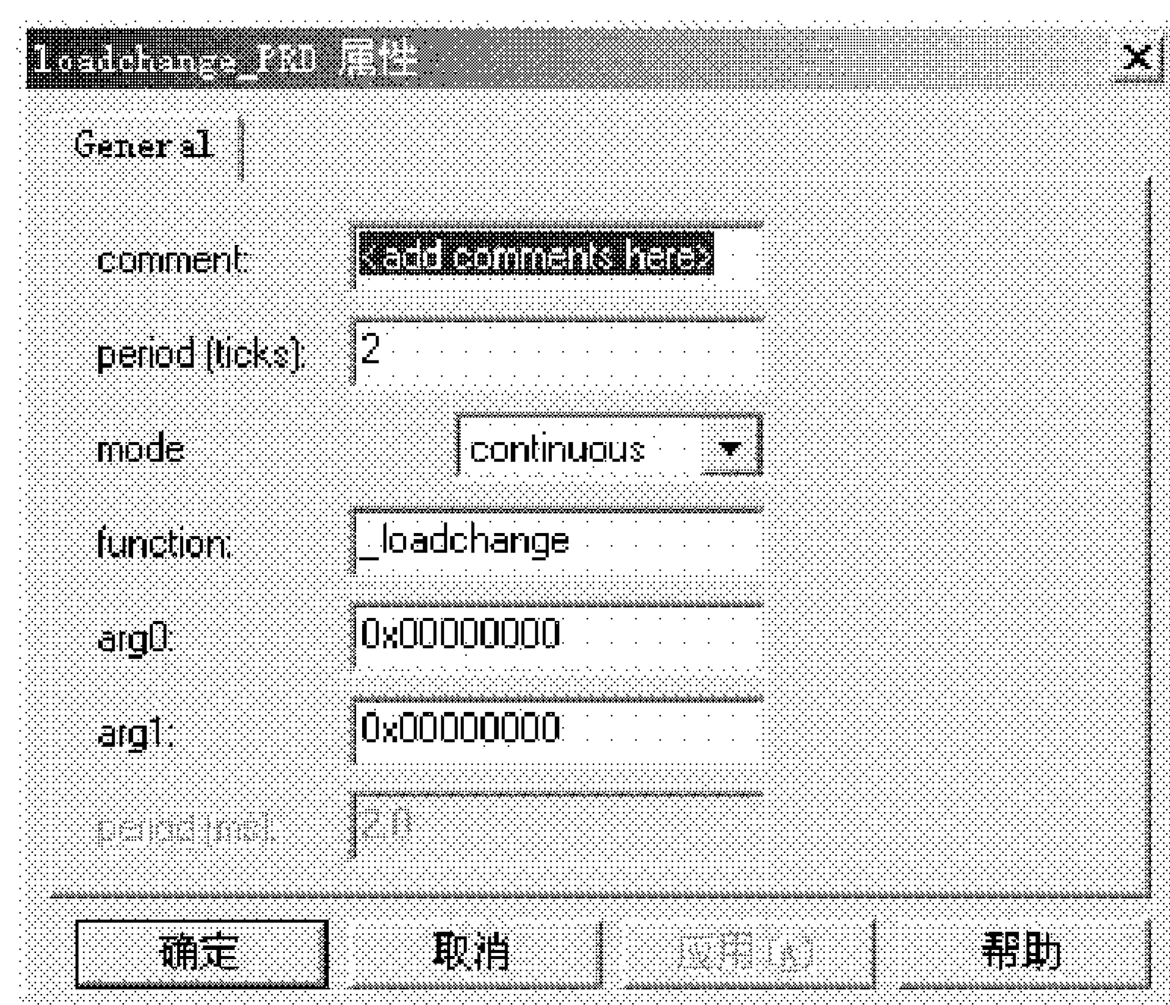


图 3.46　PRD 对象的属性对话框

• period(tcks)：此值设置 PRD 对象的周期间隔。PRD 对象的执行默认为利用 CLK Manager 来触发，而 CLK 默认每隔 1 ms 中断一次。本例中 period 值设置为 2，因此 loadchange_PRD 对象每隔 2 ms 执行一次。

• function：当 PRD 对象执行时调用此项中输入的函数。本例中，输入_loadchange(下划线表明此函数为 C 语言函数)，loadchange 为用户编写的 C 语言函数，下面再对此函数作详细说明。

• arg0, arg1：function 中函数的输入参数，由于此例中 loadchange( )函数不需要输入参数，此两项的输入都为 0。

右击 PRD－Periodic Function Manager，从弹出的菜单中选择 Properties，打开 PRD 属性对话框，确定此对话框中已选择 User CLK Manager to drive PRD 项。选择此项后，PRD 对象依靠 CLK Manager 来触发，而 CLK Manager 默认的驱动频率为 1 次/ms，因此 loadchange_PRD 对象每隔 2 ms 执行一次。我们前面说过，processing_SWI 对象每隔 10 ms 执行一次。

点击 SWI－Software Interrupt Manager 前的“+”号，可以看到 CCS 自动添加了一个 PRD_swi 对象。所有的 PRD 函数都是靠此软件中断来触发执行的。

右击 RTDX－Real－Time Data Exchange Settings，从弹出的菜单中选择 Insert RTDX，添加一个新的 RTDX 通道。给此新添加的 RTDX 通道改名为 control_channel。右击 control_channel，从弹出的菜单中选择 Properties，打开 control_channel 属性对话框，在此对话框的 Channel Mode 项中选择 input，点击 OK 退出此对话框。

(17) 保存修改过的 DSP/BIOS 配置文件 volume.cdb。

(18) 修改源代码文件。

双击工程视窗中的 volume.c 文件，在 CCS IDE 的源代码编辑窗中打开此文件，并对其进行修改。向 volume.c 源文件中添加 loadchange( )子函数，函数如下所示。

```
void loadchange(void);   /*loadchange 函数声明，放在主函数 main( )的前面*/
  /*======== loadchange ========*/
Void loadchange( )
  {
    static Int control = MINCONTROL;
     /* Read new load control when host sends it */
     if (!RTDX_channelBusy(&control_channel)) {
        RTDX_readNB(&control_channel, &control, sizeof(control));
        if ((control < MINCONTROL) || (control > MAXCONTROL)) {
            LOG_printf(&trace,"Control value out of range");
            }
        else {
            processingLoad = control;
            LOG_printf(&trace,"Load value = %d",processingLoad);
           }
        }
      }
```

说明：loadchange( )函数从 RTDX 输入通道中读入一个数值放入到 control 变量中，根据读入值的范围，决定是否把它赋给变量 processingLoad。loadchange( )函数依靠 loadchange_PRD 对象来执行，每隔 2 ms 执行一次。

双击工程视窗中的 volume.h，在 CCS IDE 的编辑窗中打开此头文件。修改 MAXCONTROL 的定义为 1900，即 #define MAXCONTROL 1900。

(19) 保存修改后的源代码文件，重新编译链接工程并加载运行。

(20) 利用主机程序向 RTDX 输入通道写入数据。

下面的一段 MATLAB 程序用于向 RTDX 输入通道写入数据，从而修改目标程序中的 processingLoad 变量值。

```
cc=ccsdsp('boardnum', 0, 'procnum', 0);       %创建 MATLAB 与 CCS IDE 的连接对象
cc.rtdx.open('control_channel', 'w');         %打开 RTDX 输入通道 conrtol_channel
cc.rtdx.enable('control_channel');            %使能 RTDX 输入通道 control_channel
cc.rtdx.enable;                               %使能 RTDX 接口
cc.rtdx.isenabled('control_channel');         %确定 RTDX 输入通道 control_channel 是否已使能，向
                                              %RTDX 通道读/写数据之前都要首先确定 RTDX 通道是
                                              %否已使能。如果 cc.rtdx.isenabled 返回 0，则需要利用
                                              %cc.rtdx.enable 函数重新使能此通道。
cc.rtdx.writemsg('control_channel', int32(200); %把 200 写入到 RTDX 输入通道 control_channel 中，
                                                %用来修改变量 processingLoad 的值，利用此方法继
                                                %续写入其它的数值。
                         ⋮
cc.rtdx.writemsg('control_channel', int32(100)); %继续向 RTDX 输入通道写入其它数据来修改变量
                                                 %processingLoad 的值。
```

(21) 随 processingLoad 值的变化观察目标程序中各线程的执行情况。

利用 RTDX 通道，修改变量 processingLoad 的值为 100，Execution Graph 窗口显示目标程序中各线程的时序关系，如图 3.47 所示。

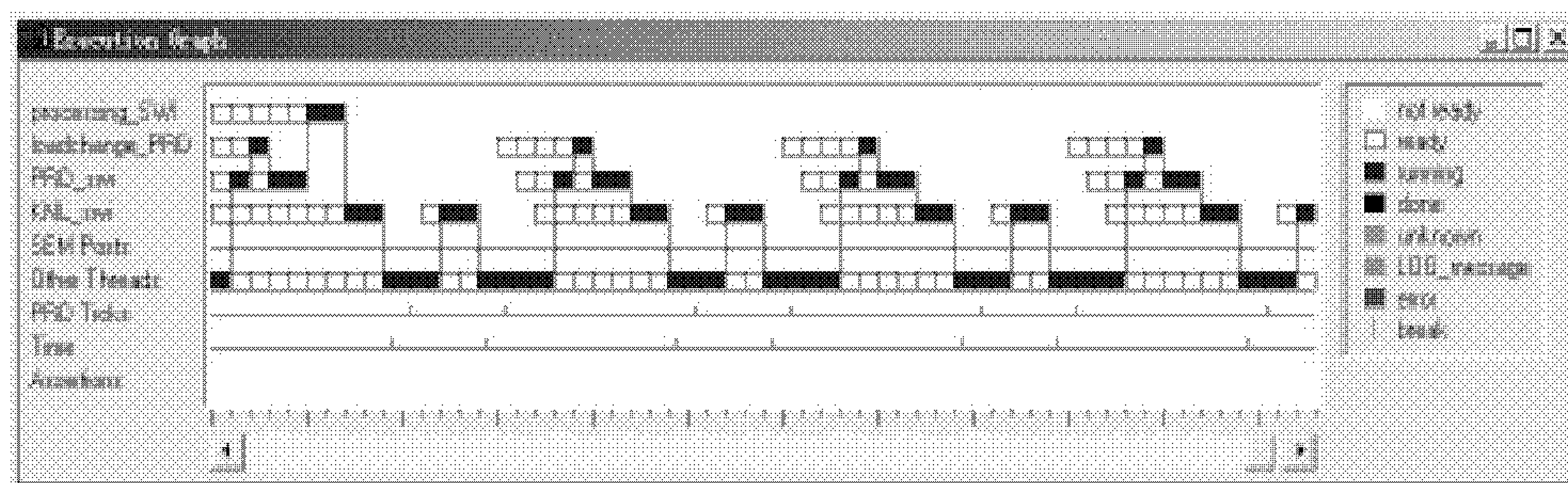


图 3.47　Execution Graph 显示的各线程执行时序(processingLoad 值为 100)

从图 3.47 可以看到，processing_SWI 每隔 10 个短竖线运行一次(10 ms)，而 PRD_swi 每隔 2 个短竖线运行一次(2 ms)，这与我们所期望的一致，因此目标程序中各线程的时序关系不存在时间问题。

继续修改变量 processingLoad 的值为 600，此时 Execution Graph 窗口的显示结果如图 3.48 所示。可以注意到 Execution Graph 窗的 Assertions 轴上出现几个红色小方块，此红色小方块表明 CCS IDE 已发现某一线程不满足时序要求。由于 processingLoad 的值过大，导致 processing_SWI 的执行时间过长(已超过 2 ms)，而 processing_SWI 与 loadchange_PRD 具有相同的优先级，在 processing_SWI 的执行过程中 loadchange_PRD 无法再执行。loadchange_PRD 要求每隔 2 ms(对应 Execution Graph 窗中 Time 轴的 2 个短竖线)执行一次，因此当 processing_SWI 的执行时间超过 2 ms 时，loadchange_PRD 不再满足期望的时序关系。

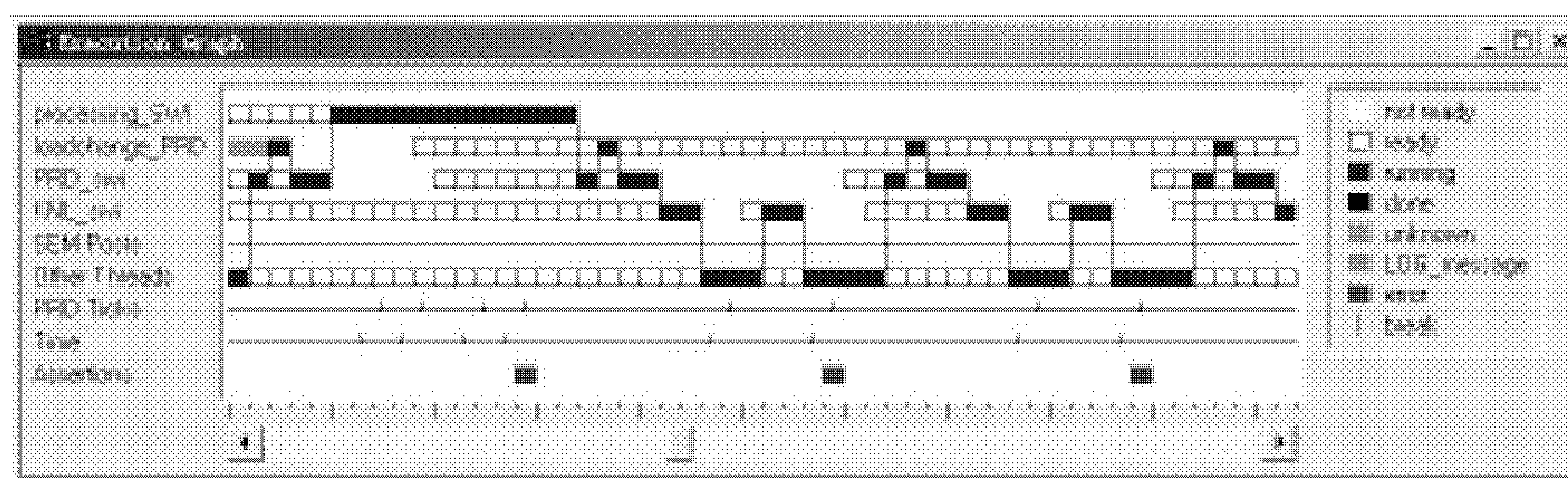


图 3.48　Execution Graph 显示的各线程执行时序(processingLoad 值为 600，存在时序问题)

(22) 修改线程的优先级。

下面通过修改线程的优先级来解决上述时序问题。

停止目标 DSP 中程序的执行。双击工程视窗中的 volume.cdb 文件，在 CCS IDE 中打开此 DSP/BIOS 配置文件。高亮选中 SWI－Software Interrupt Manager，在左边的属性显示窗中显示出 SWI 的所有优先级，如图 3.49 所示。KNL_swi 具有最低的优先级，此软件中断用来运行 TSK manager。PRD_swi 对象和 processing_SWI 对象具有相同的优先级，因此在 processing_SWI 对象的执行过程中 PRD_swi 对象就无法执行，这正是前面出现时序问题的原因。要解决这一问题，只要用鼠标把 PRD_swi 对象拖到一个较高的优先级上(例如优先级 2)就可以了。

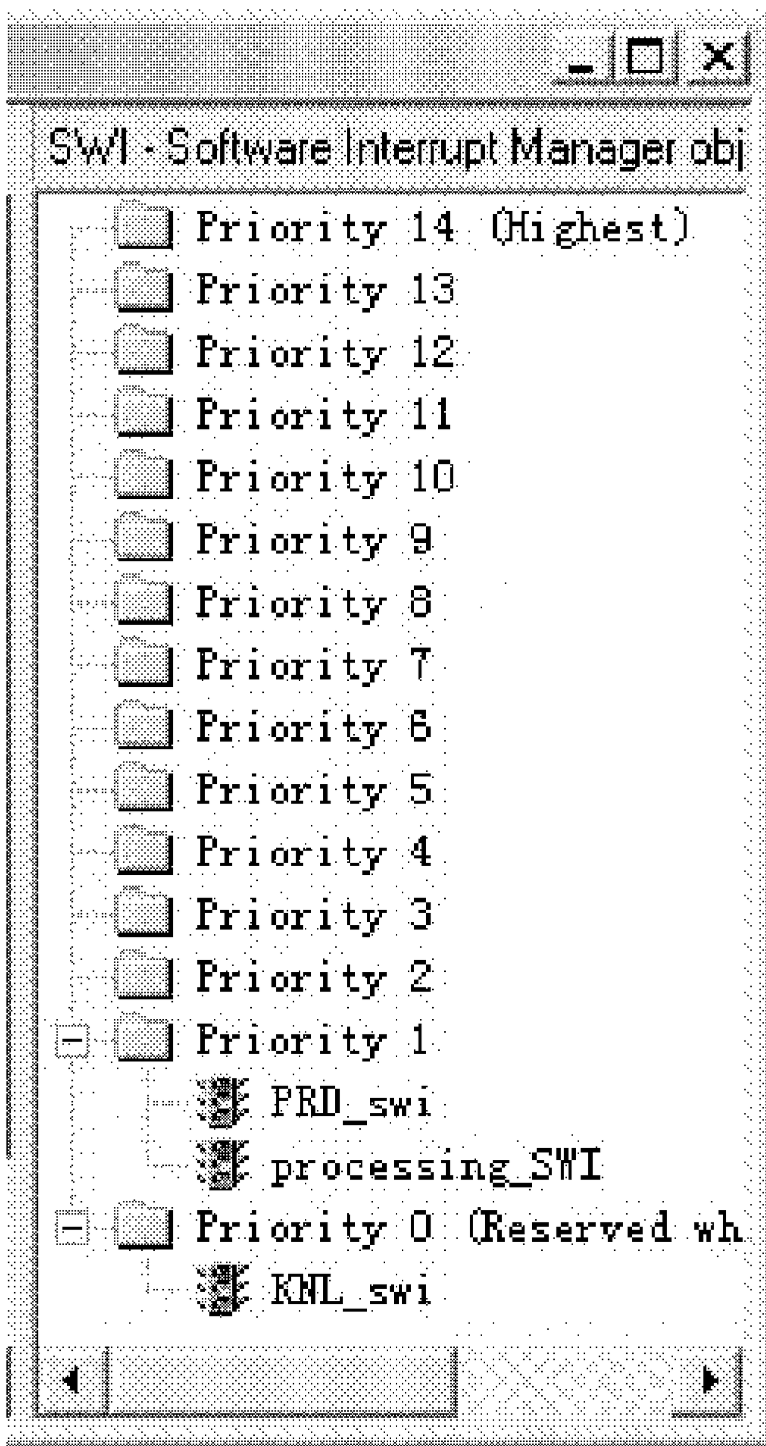


图 3.49　SWI 中的所有软件中断及其优先级

保存此修改过的 DSP/BIOS 配置文件，重新编译链接工程，并加载运行生成的可执行代码。

仍然利用 RTDX 通道修改 processingLoad 的值为 600，此时 Exeution Graph 窗口的显示结果如图 3.50 所示。比较图 3.50 和图 3.48 的结果可以看出，此时所有线程的时序关系都已经满足要求了。

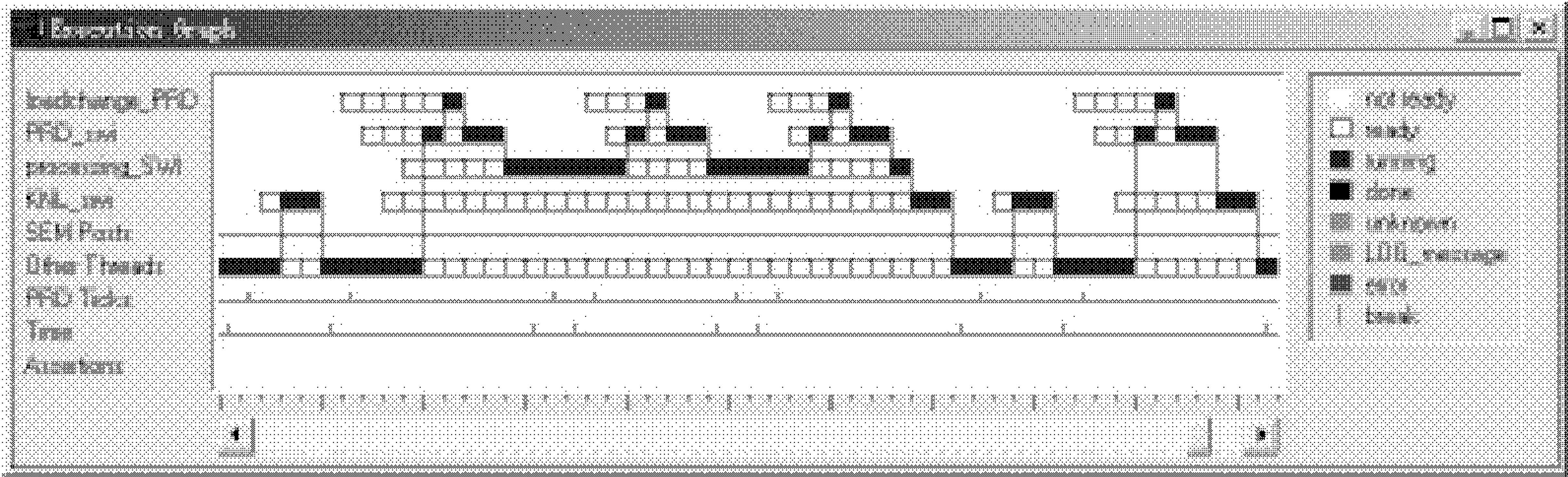


图 3.50　Execution Graph 显示的各线程执行时序(processingLoad 值为 600，满足时序要求)

(23) 停止目标 DSP 中程序的执行，关闭所有 CCS IDE 窗口。

至此，我们已完成了所有实时性调试任务。

思　考　题

3.1　CCS 是 TI 公司推出的一个集成 DSP 开发环境，可以支持 TI 公司的哪些 DSP？其操作界面是否相同？CCS 具有哪些功能？CCS 是否可以支持 TI DSP 软件开发的整个过程？

3.2　读者用过哪些其它的 DSP 开发工具？CCS 相对于其它的 DSP 开发工具增加了哪些功能或特点？

3.3　用 C/C++语言编写一个 FFT 程序，练习应用 CCS 中提供的编辑工具、代码生成工具和调试工具，最终生成可执行的目标 DSP 代码。

3.4　CCS 中的文件类型有哪些？每种文件类型的后缀和功用是什么？这些文件类型之间的关系是什么？

3.5　什么是链接命令文件？有什么用途？利用 C/C++语言混合编程和只利用汇编语言编程时的链接命令文件是否相同？如何修改？

3.6　断点(Breakpoint)和探点(Probepoint)的区别是什么？各有什么用途？如何设置条件断点和硬件断点？CCS 也提供了对存储器内容的画图工具，直接利用此画图工具与把数据输入到 MATLAB 环境中，再利用 MATLAB 的工具进行画图，哪个更方便、更好用？

3.7　CCS 是一个可供第三方接入的开放式结构，如何利用 Visual Basic 或 Visual C++ 程序通过 COM 接口来访问 CCS 中的数据？

3.8　什么是代码实时性调试工具？它与传统的调试工具有什么不同？为什么需要实时性调试？DSP/BIOS 是一种实时操作系统，它只运行在目标 DSP 中，还是只运行在主机端？DSP/BIOS 是如何获取目标程序执行的实时信息的(即实时性调试的基本原理是什么)？传统的调试工具是利用主机通过 JTAG 仿真器定时地扫描处理器资源来获得程序执行信息的，而 DSP/BIOS 是如何获得的呢？

3.9　应用 DSP/BIOS 的配置工具添加 DSP/BIOS 的调试模块时，是否需要修改目标 DSP 的程序？如何修改？

3.10　DSP/BIOS 是否也可作为应用系统的一部分，以节省软件开发周期？怎么应用？

3.11　如何发现目标 DSP 程序中存在的与时间有关的问题？如何解决时间瓶颈问题？

3.12　什么是实时数据交换技术（RTDX）？它所依靠的硬件和软件资源是什么？DSP/BIOS 实时分析工具就是利用 RTDX 作为低层通信基础来实时显示和分析目标程序执行信息的，RTDX 为什么对目标程序的执行影响很小？

3.13　目标 DSP 程序中需要用到 RTDX 功能时，如何修改其源代码？如何创建 RTDX 通道、打开 RTDX 通道、向 RTDX 输入通道中写入数据或从 RTDX 输出通道中读出数据？

3.14　如何利用 MATLAB 来产生目标 DSP 程序的测试数据？如何把 MATLAB 产生的测试数据输入到目标 DSP 中？反之又如何呢？如果利用 MATLAB 6.5 中提供的 MATLAB Link for CCS Development Tools 工具包，会带来什么方便呢？

3.15　如何利用 MATLAB 来打开 RTDX 通道、使能 RTDX 通道、向 RTDX 输入通道中写入数据或从 RTDX 输出通道中读出数据？

3.16　利用 MATLAB 的 FDATool 工具设计一个 IIR 滤波器，并把滤波器的系数输出到目标 DSP 中。考虑输入之前如何进行数据类型转换，数据类型转换是否会带来精度问题，甚至导致滤波器不稳定。

第 4 章　SHARC DSP 集成开发环境 VisualDSP++

与第 3 章介绍的集成开发环境 TI CCS 类似，AD 公司也推出了自己的集成开发环境 VisualDSP++。VisualDSP++支持 AD 公司的所有 DSP 产品，包括 SHARC、TigerSHARC、Blackfin、ADSP218x 和 ADSP219x DSP。VisualDSP++的界面、功能及其操作方法与第 3 章介绍的 TI CCS IDE 非常类似。本章以 VisualDSP++3.0 为例，详细介绍如何对 SHARC DSP 的程序进行开发调试。

4.1　VisualDSP ++开发工具概述

VisualDSP++是一个集成开发环境，开发人员在此环境下就可以完成从源代码编写、可执行代码生成到实时调试的几乎整个 DSP 软件开发过程。VisualDSP++环境下的程序开发流程如图 4.1 所示。

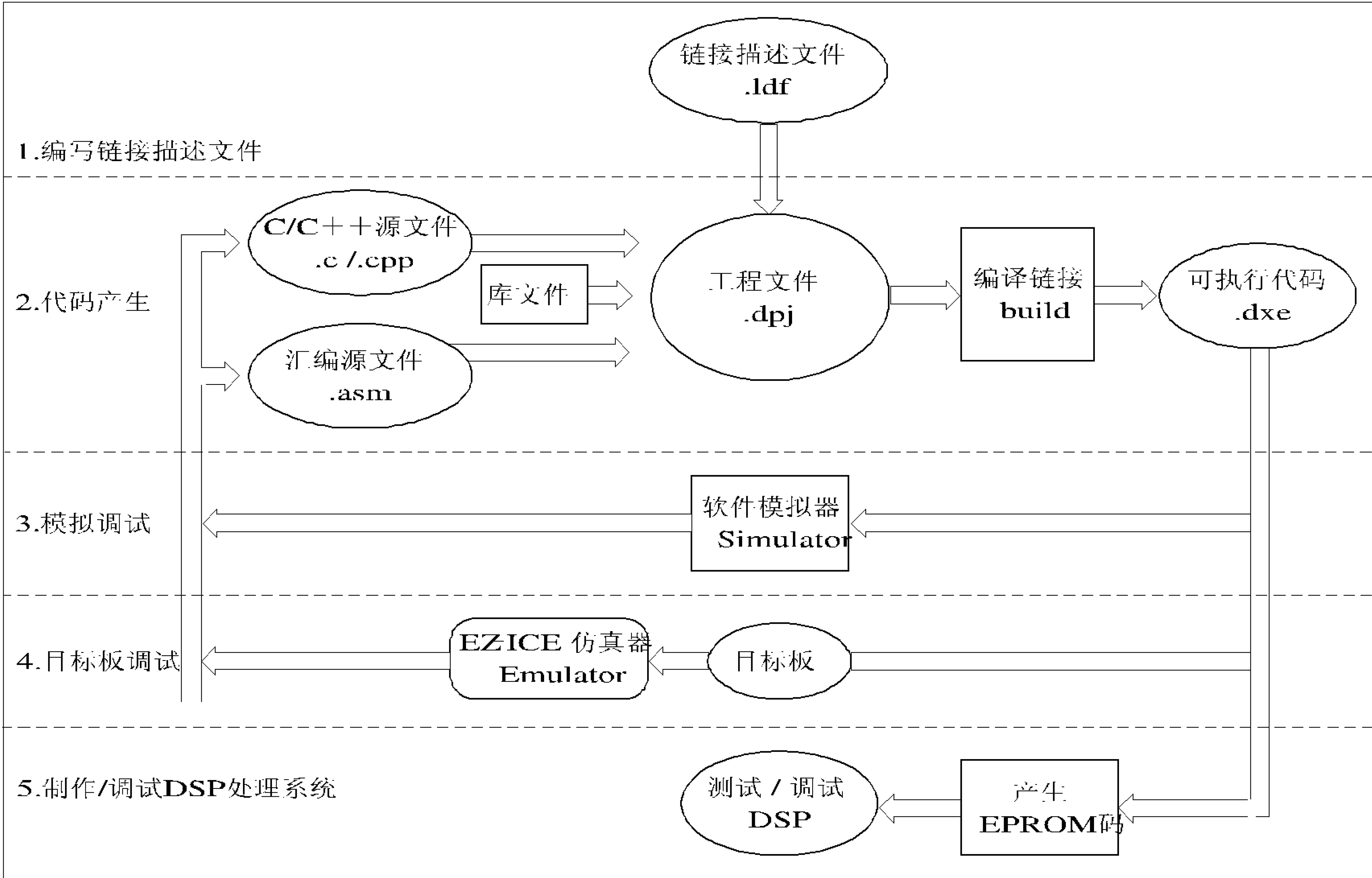


图 4.1　VisualDSP++环境下的开发流程

VisualDSP++采用工程方式管理应用程序中的所有文件，而且提供开放式接口，主机的其它应用程序(例如 MATLAB，关于 MATLAB 和 VisualDSP++的接口问题，用户可以登录到 www.sdltd.com/dspdeveloper 得到详细资料)可以通过此接口访问 VisualDSP++中的数据，进而对其分析处理。VisualDSP++的主要特点如下：

- 灵活的工程管理功能。

VisualDSP++采用工程方式管理所有文件，工程(.dpj)中包含源文件(汇编、C/C++)、链接描述文件(.ldf)、数据文件、头文件和库文件等，并可以随时添加、删除和修改文件。

- 通过图形界面来访问 VisualDSP++中集成的所有工具。

Visual DSP++通过对话框和菜单的形式来提供所有开发工具的入口参数，而且可以设定对某个文件的操作或对整个工程的操作。VisualDSP++中集成的工具包括：C/C++编译器、汇编器、链接器、加载器、软件模拟器、硬件仿真器、分割器(Split)、加载器和运行时间库等。软件模拟器(Simulator)和硬件仿真器(Emulator)具有统一的界面。

- 强大的编辑功能。

可对 C/C++源代码和汇编源代码进行编辑，编辑器能够自动识别关键字、注释等并可以设置为不同颜色进行高亮显示。支持拖拉、书签和其它标准的编辑操作。

- 灵活的代码产生(Build)工具。

可以编译链接某些文件，或对整个工程进行编译链接，或只对修改过的文件进行编译链接。输出窗中会显示编译链接的过程信息，如果过程出错，双击出错信息就会自动打开出错文件，并且光标停在出错行。

- 多语言支持功能。

VisualDSP++支持 C/C++和汇编语言程序调试。对 C/C++编写的程序，可以查看 C/C++源程序及其反汇编程序，或混合显示，即在每行 C/C++程序后显示其对应的反汇编代码。可以显示其局部变量或表达式的值。

- 完善的程序控制功能。

可以在源文件的行、标号或地址上设置断点，可以在寄存器、堆栈或存储器的位置上设置条件断点以判定何时进行了访问。

可进行单步执行(Step Into)、运行至光标处(Run to Cursor)、从函数体跳出(Step Out)、跳过函数体(Step Over)、复位 DSP(Reset)、复位程序计数器 PC 到中断矢量表的第 1 个地址处(Restart)等操作。

- 强大的代码调试功能。

可以在编辑窗口中的源代码行、反汇编窗口中的地址或程序标号处设置断点或条件断点，当程序运行到断点时停止执行；还可以在寄存器、堆栈和存储器的位置上设置监视点(Watchpoint)，当 CPU 访问这些寄存器、堆栈和存储器内容时停止程序执行。

提供的跟踪(trace)、代码性能统计(profile)和 Statistical profile 等功能，能够迅速发现 DSP 程序中的瓶颈现象及需要优化的程序块。

能够模拟中断、I/O 数据传送等真实应用环境。

在程序运行过程中能够查看寄存器/存储器中数值的变化，还可以同时在编辑窗口或反汇编窗口中观察源代码的执行过程，甚至可以观察 Cache 和流水线的执行情况。

能够对存储器中的一段数据进行图形显示，包括 plot、星图、眼图，甚至三维图等，而

且可以对数据进行对数显示(dB)以及显示数据的谱(FFT 幅度)，对图形的操作方便、灵活。

- 具有同时多 DSP 调试能力。

在一个界面下能够同时调试任意数目的 DSP。具有多 DSP 同步操作功能，如同步 Step (单步执行)、同步 Run (连续执行)和同步 Halt(停止)等。

与 VisualDSP++配套的硬件仿真器有以下几种：

SUMMIT－ICE：PCI 插卡；

APEX－ICE：USB 接口；

Trek－ICE：网络接口。

4.2 VisualDSP ++的代码产生工具

VisualDSP++集成开发环境对 ADSP 应用程序开发的几乎整个过程提供支持，在 VisualDSP++中所有应用程序的开发一般都要经过如下几步：

(1) 创建一个新工程。

VisualDSP++利用工程来管理整个应用程序中的文件，因此开发一个 DSP 应用程序之前，首先要创建一个工程文件(.dpj)。

(2) 设置工程选项(Project Options)。

创建一个工程文件后，还要设置工程选项，包括指定目标 DSP 类型，指定生成可执行文件的类型、名称和路径，设置 C/C++编译器、汇编器、链接器、分割器(Splitter)和加载器(Loader)的选项等。

(3) 添加或编辑工程源文件。

向工程中添加已编辑好或生成的源代码文件、头文件、数据文件和库文件等，或直接利用 VisualDSP++的编辑工具进行编辑。

(4) 编译链接(Build)工程。

VisualDSP++提供了两种基本 Build 配置：

- Debug 版：编译链接生成用于调试的可执行代码，即指示编译器、汇编器生成调试信息等，称为调试版的代码。
- Release 版：编译链接生成最终加载的可执行代码，称为正式版的代码。

因此应首先选择 Debug 工程配置，编译链接生成可执行代码后才能利用 VisualDSP++的调试工具进行调试。

当然用户也可以指定自己的编译链接选项，即设置 C/C++编译器、汇编器、链接器选项。

(5) 调试工程。

把可执行代码加载后，再利用 VisualDSP++提供的强大的调试工具进行调试，快速发现程序中存在的问题。

(6) 编译链接生成正式(Release)版的代码。

当程序调试完成后，重新选择 Release 工程配置，重新编译链接工程，生成正式版代码。

选择 Start(桌面开始菜单)→Programs→VisualDSP→VisualDSP++ for SHARC(应用于

SHARC DSP 的 VisualDSP++)，打开 VisualDSP++主界面，如图 4.2 所示。

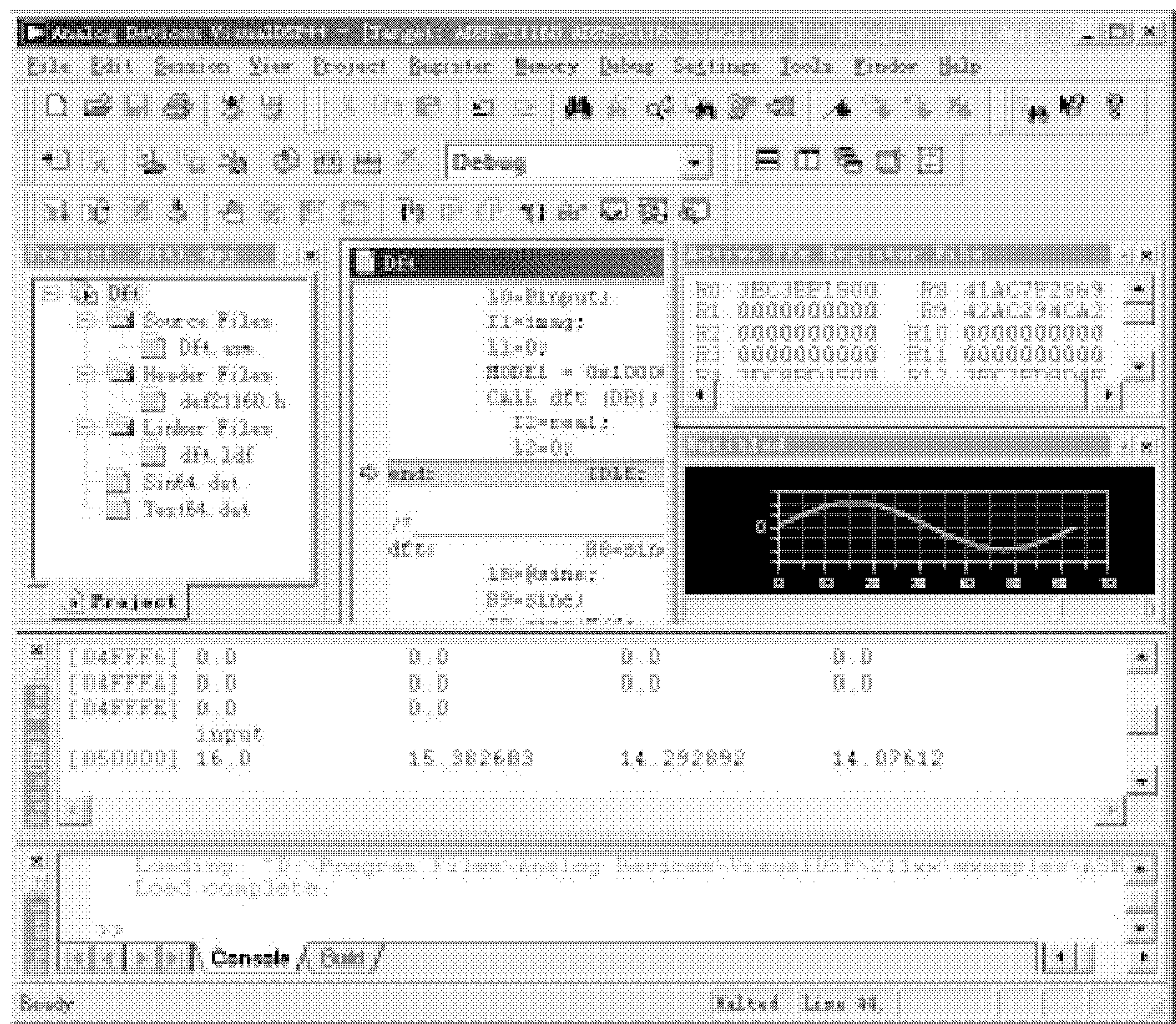


图 4.2　VisualDSP++的开发界面

下面对以上程序开发过程分别进行详细介绍。

1. 创建一个新的工程文件

在 VisualDSP++环境下，DSP 的所有应用开发都是基于工程的，所以创建一个工程文件是整个应用软件开发的第一步。工程文件(*.dpj)中存放程序的编译链接信息：源文件列表、关系(Dependencies)信息和开发工具的选项设置等。

(1) 选择 Start(桌面开始菜单)→Programs→VisualDSP→VisualDSP++forSHARC，打开 VisualDSP++主界面。

(2) 选择 Project→New，打开一个 Save New Project as 对话框。

(3) 在 Save New Project as 对话框中指定路径并输入工程名，点击 Save 保存此新工程，同时打开工程选项对话框(Project Options)，如图 4.3 所示。在 Project 面板中设置如下选项：

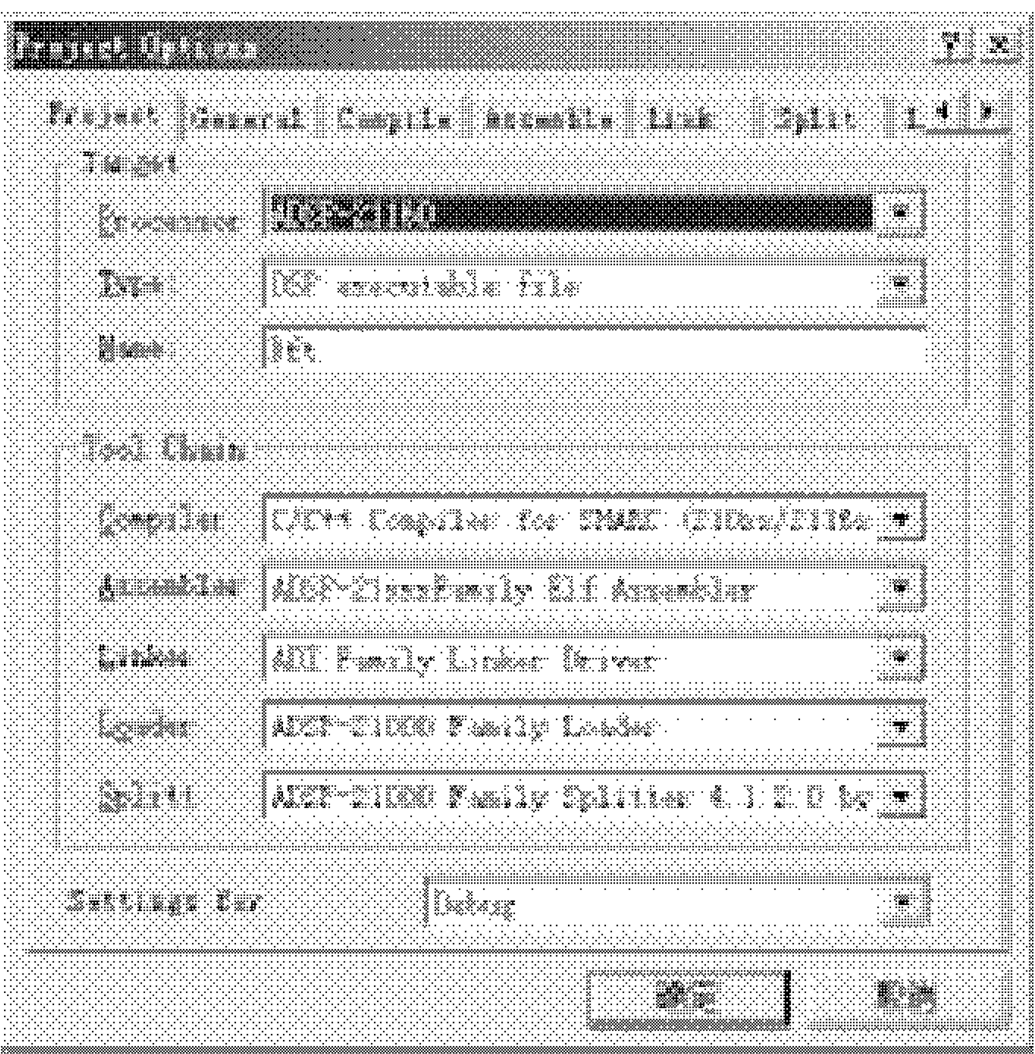


图 4.3　工程选项对话框

• Processor：选择 DSP 的类型(如 ADSP－21160、ADSP－21060、ADSP－21062 等)。

• Type：选择 Build 输出文件类型，有以下几种：

Archive file：生成由.DOJ 文件组成的档案库文件(.DLB)。

DSP executable file：生成一个可执行文件(.DEX)，此文件可以加载到 DSP 目标系统中进行调试。

DSP object file：生成一个可重新定位的目标文件(.DOJ)。

Loader file：产生一个工业标准的引导加载文件(.LDR)，在内部存储器中执行。

Splitter file：产生一个非加载的 PROM 镜像文件，直接在外部存储器中执行。

• Name：指定一个输出文件名。

• Settings for：选择 VisualDSP++为工程提供的两种基本 Build 配置：Debug 或 Release。

关于 Project Options 对话框中的其它工具选项设置，我们在后文中再详细介绍。

(4) 设置好 Project Options 对话框后，点击 OK 退出 Project Options 对话框，同时会出现一个询问是否在此新工程中应用 VDK(VisualDSP++ Kernel)的对话框。VDK 是 VisualDSP++2.0 以后新增加的一大功能，类似于 CCS 中的 DSP/BIOS 功能，它是一种运行在目标 DSP 上的实时操作系统。用户可以把 VDK 模块添加到应用程序中，用来配置外设、设置线程优先级、实时任务调度、中断、I/O 服务和其它实时操作等。点击 No，退出添加 VDK 对话框。

(5) 在 VisualDSP++的工程视窗中可以看到新创建的工程，但此时工程中不包含任何文件。

2．向工程中添加或编辑源文件

工程中可以包含一个或多个 C/C++语言、汇编语言源文件、头文件和数据文件等。当创建了一个工程文件并在工程选项中指定了所用的 DSP 类型后，就可以把新的或已编辑好的源文件加入到该工程中去。

(1) 添加文件到工程中。

可以把任何类型的文件添加到工程中去，当进行编译链接时，VisualDSP++能自动选择可识别的文件进行编译链接。

选择 Project→Add to Project→File(s)，或右击工程视窗中的工程名，从弹出的菜单中选择 Add File(s) to Project，指定路径和文件名，然后点击 Add 把此文件添加到工程中。开发人员还可以在工程视窗中添加一个文件夹。

(2) 编辑一个源文件并把它加入到工程中。

选择 File→New，或点击工具栏中的编辑新文件图标，打开源文件编辑窗口，接下来就可以进行编辑了。VisualDSP++提供的编辑功能是非常强大的，不但支持标准的编辑功能，还支持用户将语法按不同颜色高亮显示，还可以加入书签和进行列编辑操作等。

选择 Settings→Preferences，打开 Preferences 对话框，其中的 Editor 面板设置文本编辑操作：

File type：选择 C/C++语言、汇编语言、链接描述文件或其它类型的文件。

Tab size：设置 Tab 的大小，即每按一次 Tab 键退后的空格数(1～64 的整数)。

Show tabs：使 Tab 的特征字在文中显示出来。

Insert space：用空格来替代 tabs。

Keep tabs：保持 tab 特征字。

Coloring：点击进入 Coloring，可以对语法设置不同颜色高亮显示。

Font：点击进入 Font，可设置字体。

其它一些标准的编辑操作，如 copy、paste、cut 和书签等功能与其它的编辑器一样。把新编辑的源文件存盘，并加入到工程中去。

(3) 工程中的文件关系信息(Dependencies)。

Dependencies 用来描述工程中源文件之间的关系，即哪一个文件需要用到另一个文件的信息，因此这决定了编译链接的顺序。VisualDSP++的工程管理工具保存 Dependencies 信息。如果改变了 Build 选项或加入一个新文件，VisualDSP++就需要更新 Dependencies。更新工程 Dependencies 可以通过选择 Project→Update Dependencies 来实现。在编译链接期间，会更新文件的 Dependencies 信息。

查看文件的 Dependencies 信息：右击工程视窗中的文件名，从弹出的菜单中选择 Properties，在 Properties 对话框的 Dependencies 栏中查看此文件的关系信息。

3．设置工程 Build 选项

我们在前面提到过，VisualDSP++为工程提供了两种基本 Build 配置：Debug 和 Release。在前面创建新工程时我们已经选择了 Debug 配置，在 VisualDSP++主界面上会显示此配置，如图 4.4 所示，也可以直接从图 4.4 显示的下拉菜单中重新选择工程配置。



图 4.4　显示工程 Build 配置

用户除了利用 VisualDSP++提供的基本 Build 配置外，还可以设置自己的工程 Build 选项，或对工程中的单个文件设置编译选项。

选择 Project→Project Options，打开 Project Options 对话框，如图 4.3 所示。Project Options 对话框的 Project 栏中选项，我们在前面创建新工程时已作了说明，这里不再重述。下面对 Project Options 对话框的其它面板栏中的选项分别进行介绍。

(1) General 栏中选项：

- Intermediate：指定中间结果路径。
- Output files：指定输出结果路径。

(2) Compile 栏中选项：

① 在 Category 中选择 General(一般类操作)：

- Enable optimization：允许 C/C++编译器进行优化，尤其是优化程序的循环。
- Interprocedural optimization：对整个应用程序中的源文件统一进行优化。
- Generate debug information：输出 Debuger 工具中需要的符号和其它信息。
- Stop after Preprocesser：C/C++预处理后就结束(不用进行编译)。
- Stop after Compiler：C/C++编译后结束，从输出的可执行文件中移去调试信息(符号列表和其它项)。
- Disable built－in functions：仅认可内部固有的函数(这些函数前加两个下画线)。

② 在 Category 中选择 Preprocessor(预处理控制)：

- Definitions：定义宏名称。如果省略名称，编译器定义宏的名称为 1。通过分号相隔，可以定义多个宏。

• Undefines：取消宏的定义。通过分号相隔，可以取消多个宏。

• Additional include directories：指定路径搜索头文件。通过分号相隔，可以指定多个路径。

• Ignore include directories 选择此项，则表示只在当前路径和 Additional include directories 指定的路径中搜索头文件。

③ 在 Category 中选择 Warning(警告操作)：

• Implicit function declarations：选择此项，表示当没有明确的函数声明时会发出警告。

• Functions not inlined：选择此项，表示当函数被声明为 *inline*，但编译器不能产生 *inline* 码时会发出警告。

• Enable remarks：产生诊断信息。

• Disable all warnings and remarks：禁止所有警告信息。

④ 在 Category 中选择 DSP Specific(DSP 指定操作)：

• Reserve registers：加入编译器禁止使用的寄存器名，寄存器名之间用分号相隔。这些寄存器限于：B0, L0, M0, I0, B1, L1, M1, I1, B8, L8, M8, I8, B9, L9, M9, I9，USTAT1，USTAT2。注意，寄存器 L 和同序号的寄存器 I 必须同时禁止。

• Double size：指定 Double 类型数据的长度：64 位或 32 位，默认使用 32 位单精度存储格式。

• Additional Options：在此文本输入框中添加其它编译器命令行选项。

(3) Assembler 栏中选项：

• Preprocess only：执行完预处理就停止，没有把源文件汇编成 DOJ 文件。

• Assemble only：把源文件汇编成 DOJ 文件，跳过预处理。

• Both：上述两种操作都执行。

• Generate verbose output：把执行信息显示到标准的输出设备上。

• Generate debug information：产生行数和 DWARF－2 格式的符号信息等。

• Defines：为预处理定义标识符。

• Include Path：指定搜索路径名，这些路径中含有#include 命令包含的头文件(.h)。路径名之间用分号作间隔，可指定多个路径。

• Output listing file：产生输出列表文件.lst，其中包含指令地址、指令机器码等信息。

• Additional Options：在此文本输入框中添加其它汇编器命令行选项。

(4) Link 栏中选项：

• Search directories：加入链接器搜索的文件路径，路径之间用分号相隔。

• Generate symbol map：输出一个关于符号使用情况的链接 map 文件。

• Generate trace output：在链接器处理过程中把 DOJ 文件名输出到标准输出设备上。

• Warn once on undefined symbol：对每一个未定义的符号只警告一次。

• Strip debug symbols：从输出文件中把 debug 符号信息删去。

• Strip all symbols：从输出文件中删去所有的符号信息。

• Preprocessor include path：加入链接器搜索的头文件路径，路径之间用分号相隔。

• Additional Options：在此文本输入框中添加其它链接器命令行选项。

(5) Split 栏中选项：

Split 仅用于在外部存储器中运行的程序，而在内部存储器中运行的程序应用 Loader。

- PM ROM：从可执行码中抽出声明为 PM 区 ROM 的程序段。
- DM ROM：从可执行码中抽出声明为 DM 区 ROM 的程序段。
- Include RAM：当为输出存储器映射文件抽出信息时，从可执行码中抽出任何声明为 RAM 的程序段。
- Include ROM：当为输出存储器映射文件抽出信息时，忽略可执行码中任何声明为 ROM 程序段。
- Segments：从可执行码中抽出存储器段内容，在框内输入段名称。
- Format：选择非引导加载的 PROM 映射格式。
- PROM Width：选择 PROM 文件的宽度(用位表示)。
- Set flag to (仅用于 Byte - Stacked 格式)：在框内写入一个数字(整数)，Split 把这个数字加在 Byte - Stacked 格式文件的用户标志域中。
- Output file：指定 Split 的输出文件名，如果不指定，名称默认为 *executable.ext*，*ext* 与输出格式有关。

(6) Load 栏中选项：

- Loader：产生引导加载的文件(.LDR)。Split 仅用于在外部存储器中运行的程序，Loader 用于在内部存储器中运行的程序。
- Loader21k：把本栏的选项设置应用到加载器中。
- Mem21k：禁止加载器。对存储器进行初始化。
- Kernel file：加入与 Boot type 中指定的引导类型相对应的核函数。核函数与可执行文件一起生成后缀为 LDR 的引导加载文件。核函数放在 LDR 文件的前段，然后是 DXE 文件的初始化数据和程序代码。
- Output file：指定 Loader 的输出文件名。如果没有指定，默认文件为 *sourcefile*.ldr。
- Boot type：选择引导加载模式。PROM 为 EPROM 加载，HOST 为主机加载，LINK 为链路加载。

引导加载模式是指 DSP 不能直接运行处理程序，需要一段引导加载代码在复位后先对系统初始化，再调入实际运行的处理程序。这段引导加载代码自身需要通过某种复位后，由 DSP 自动进行的操作来调入到 DSP 片内某段特定程序存储段内，并在这种调入结束后立即开始执行。PROM/HOST/LINK 三种模式按各自不同的方式完成这种调入，而其后的系统初始化和调入实际处理程序的过程大同小异。在上述过程完成后，这段引导加载代码自动进行自我覆盖以保证存储器的充分利用，然后跳转到实际处理程序代码，这时的处理程序代码和数据与 DXE 文件的内容完全一致。Non-boot(非引导加载模式)远没有这么复杂，程序放在 800000h(对于 ADSP21160)或 400000h(对于 ADSP2106x)开始的外部存储器中，而复位后程序计数器 PC 一定指向这一起始地址。

- Hex start address (仅用于 PROM 加载模式)：指定十六进制的 EPROM 地址，在这个地址处开始引导加载文件。
- Mutiprocessor：表示多 DSP 输出。在文件列表框内加入每一个 DSP 的可执行文件名。
- Format：选择引导加载文件的格式。

• Timeout：指定最大总线锁定时间，用于规定HOST引导时总线锁定时间不超过2N个时钟周期。N取3～32 765之间的整数。

• Verbose：在Loader执行过程中显示状态信息。

• Additional Options：在此文本输入框中添加其它Loader操作。

4．编译链接(Build)一个调试(Debug)版本的工程

选择Project→Build Project或点击工具栏中的Build Project图标来编译链接前面创建的工程。

在编译链接过程中，输出窗中会显示状态信息。如果出错，会显示出错信息。用鼠标双击出错信息行，会自动打开出错的源文件并且光标停在出错行。

如前所述，输出文件类型(工程选项中)必须指定为DSP executable file类型(*.dxe)，才能产生可进行Debug调试的输出文件。

5．加载可执行代码

选择File→Load Program或点击工具栏中的Load图标来把生成的可执行代码加载到Simulator或目标DSP中。

6．调试(Debug)工程

把一个工程编译链接生成可执行代码并且加载后，接下来就可以用VisualDSP++中的调试工具来调试该工程了。

关于VisualDSP++中的调试工具，在4.3节中再作详细介绍。

7．编译链接(Build)一个Release版本的工程

当DSP应用程序完成调试后，最后一步就是重新生成最终在目标DSP上运行的可执行代码，即删除所有符号信息。经过下面的步骤来编译链接一个Release版本的工程：

(1) 在VisualDSP++的界面中选择Release工程配置；

(2) 重新编译链接这个工程。

4.3 VisualDSP++的调试工具

VisualDSP++中集成了强大的调试工具，包括完善的程序控制功能、断点、条件断点、寄存器和存储器观察窗口、C/C++变量观察窗口、统计代码段的执行性能、对存储器内容进行图形显示、观察堆栈使用情况等，利用这些调试工具可以快速发现程序中存在的问题。本节详细介绍这些调试工具。

1．源代码窗口操作

源代码窗口用来编辑源代码及对源文件进行显示，包括汇编文件(.asm)、C/C++语言文件(.c / .cpp)、数据文件(.dat)、头文件(.h)、链接描述文件(.ldf)等，还可以混合显示C/C++源程序，即在每一行C/C++语句后显示其对应的反汇编代码。

2．反汇编窗口操作

在反汇编窗口中可以查找字符、跳转(Ctrl+G)到某一指定地址上、设置断点，甚至还可以修改指令。反汇编窗口中最左边的字母A、D、F和符号=>指示当前程序的执行状态，其

含义分别为：A(放弃)、D(解码)、=>(执行)、F(取指)。

3．程序执行控制

VisualDSP++的程序运行命令在 Bebug 菜单下或工具栏中。MP(多 DSP)程序运行命令在 Bebug 菜单下的 Multiprocessor 中，MP(多 DSP)与 SP(单 DSP)的命令操作相同，只不过在 MP 下是对组中所有 DSP 执行同样的操作。

Run：运行程序直到某种情况停止它，如执行到断点或用户干预时。当停止时，所有的窗口内容都更新到当前值。

Halt：停止程序执行。当停止时，所有的窗口内容都更新到当前值。状态条显示当前程序停止的地址。

Run To Cursor：程序执行到光标所在位置处。可以在源文件窗或反汇编窗中置入 Cursor。

Step over：执行 1 行程序，跳过函数体。仅用于 C/C++语言程序。

Step Into：单步执行程序。每执行 1 步窗口都更新。

Step Out Of：跳出当前函数体，直到返回到它的调用程序。仅用于 C/C++语言程序。

Reset：复位到某一确定状态，如果与硬件相连，Reset 相当于 DSP 的 RESET 管脚输入，必须重新加载程序。

Restart：使程序计数器 PC 跳到中断矢量表的第 1 个地址处。

4．断点(Breakpoint)

当程序运行到断点时，会停止程序的执行，从而可以查看程序的运行状态，查看当前寄存器和存储器中的内容，观察堆栈调用情况等。断点可以在编辑窗口中源代码的某一行、反汇编窗口中的地址或程序标号处设置。

把光标停在源代码或反汇编代码的某一行，然后点击工具栏中 Toggle Breakpoint 图标，或右击鼠标从菜单中选择 Insert Breakpoint，在此行处设置非条件断点。

也可以设置条件断点，即当指定的条件为真时，才在断点处停止程序的执行，否则继续执行。设置条件断点：选择 Settings→Breakpoints，打开 Breakpoints 对话框，如图 4.5 所示。Breakpoints 对话框中列出了所有已设置的断点。

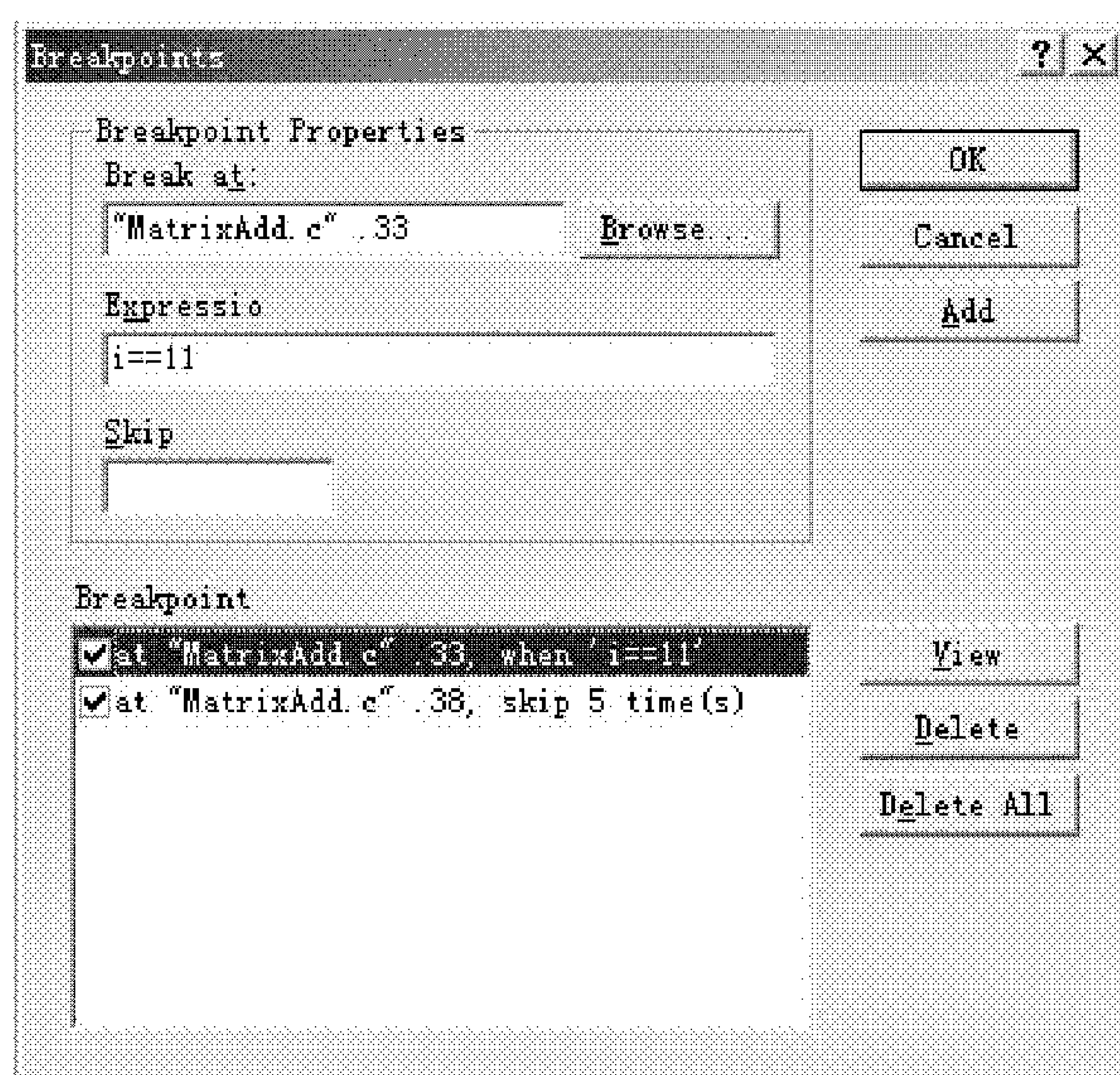


图 4.5　Breakpoints 对话框

Break at：设置断点位置。在此项中输入源代码的某一行或反汇编窗的某一地址。

Expression：输入一个表达式，当表达式的值为真时，在此断点处才停止执行。

Skip：输入一个数字，当经过断点的次数超过这一数字后，才停止程序的执行。

如果 Expression 和 Skip 都为空，则此断点为非条件断点，即每次到达此断点时都会停止程序的执行。点击 Add，添加设置的断点。

5．监视点(WatchPoint)

Watchpoint 与 Breakpoint 功能非常相似，可以在程序的各个位置处停止其执行。而 Watchpoint 可以通过设置硬件条件来停止程序的执行，包括存储器/寄存器访问和堆栈弹出等。

通过下列步骤来设置 Watchpoint：

(1) 选择 Settings→Watchpoints，打开 Watchpoints 对话框，如图 4.6 所示。

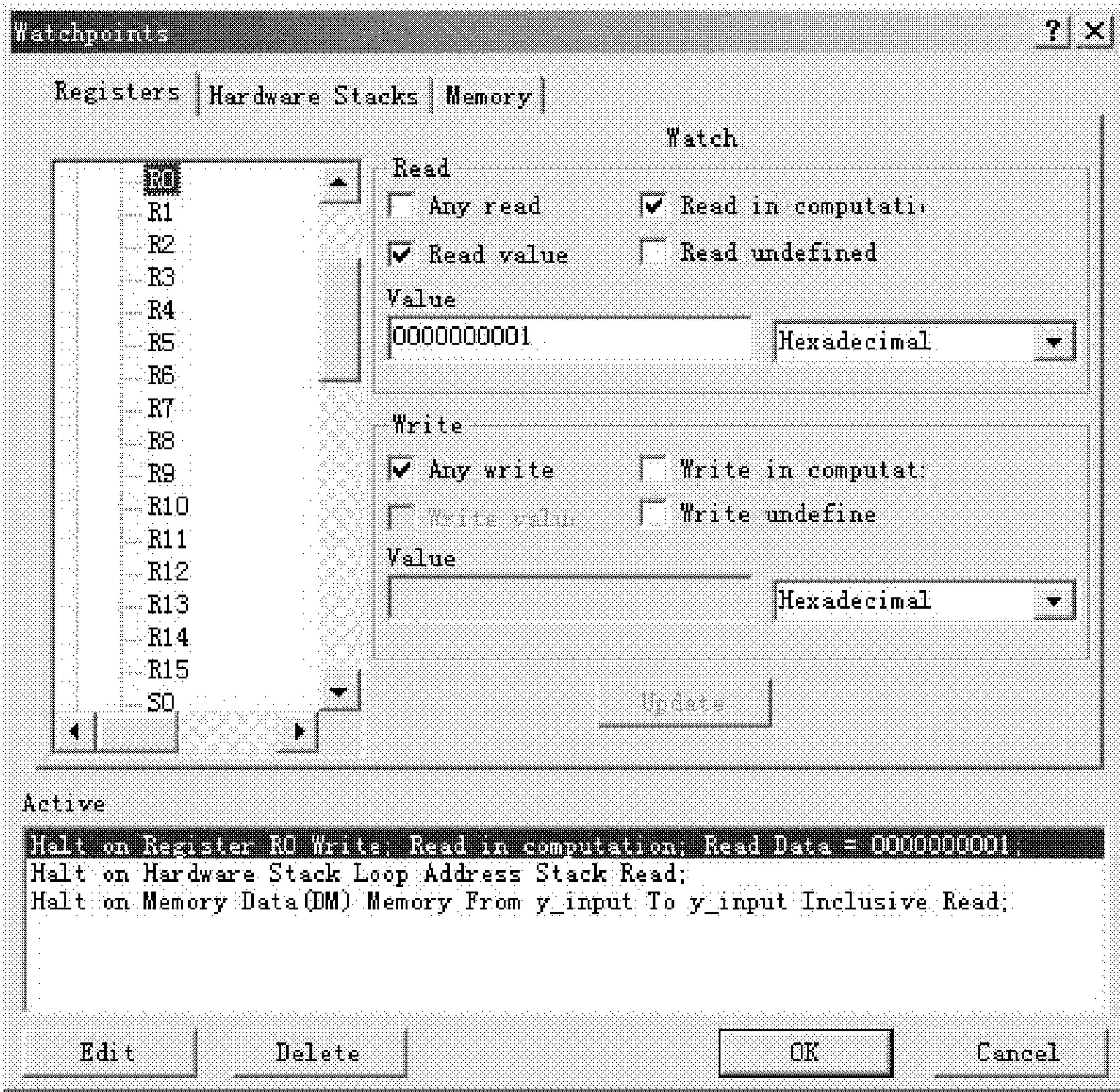


图 4.6　Watchpoints 对话框

(2) 在此对话框中指定程序停止的条件，其各部分的描述如下：

Registers/Memory：当从寄存器/存储器单元中读或向寄存器/存储器单元中写任意值或指定值时，就停止程序的执行。对于寄存器访问还可以选择是否为计算指令访问。

Hardware Stacks：当堆栈弹出或压入任意内容或指定值时，就停止程序的执行，还可以监视堆栈是否溢出。

(3) 点击 Add，添加一个监视点。

(4) 点击 OK，退出 Watchpoints 对话框。

6．寄存器观察窗口

利用寄存器观察窗口来查看当前寄存器中的内容。选择 Register → Core(核寄存器)或 IOP(外设寄存器)，打开寄存器观察窗口，如图 4.7 所示。

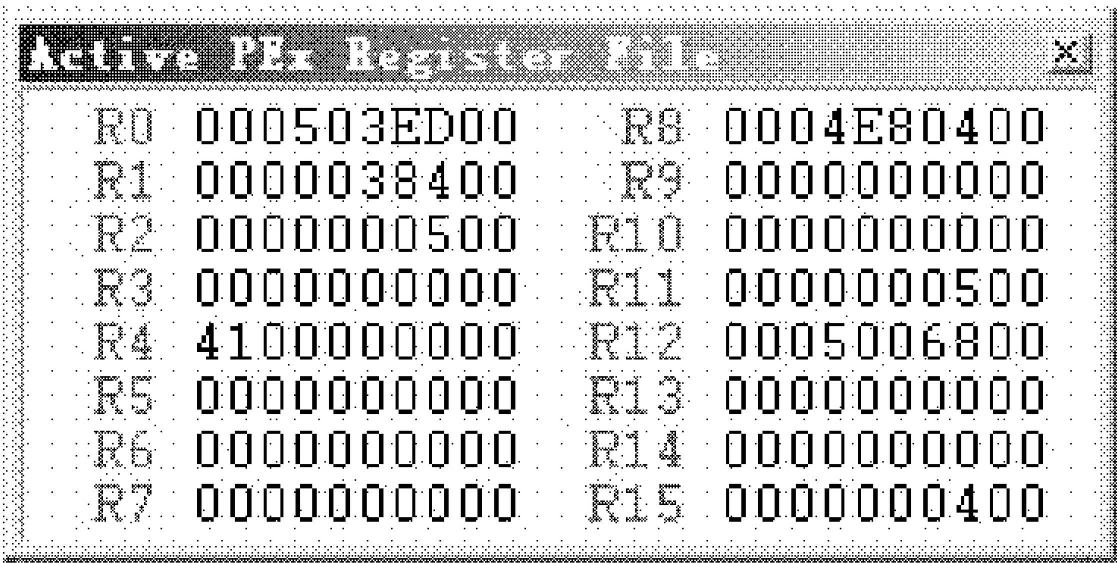


图 4.7　寄存器观察窗口

可以改变寄存器内容显示的数据格式和修改寄存器内容。寄存器显示的数据格式包括：十六进制、八进制、二进制、符号或无符号整数、32/40 位浮点、符号或无符号小数等。

改变寄存器数据格式：在寄存器观察窗中右击鼠标，从弹出的快捷菜单中选择相应的数据格式就可以了。

修改寄存器内容：在寄存器观察窗中双击要修改的寄存器值，当它高亮显示后，输入新值并回车就可以了。也可以用编辑操作，如 Edit、Cut、Paste 等对寄存器值进行修改。

7．存储器内容显示

存储器显示窗口用来显示一段存储器内容。选择 Memory → Short Word(短字格式)、Two Column(正常字)、Long Word(长字)或 Three Column(指令字)，打开存储器显示窗口。

存储器显示窗口不但像寄存器窗口那样提供不同的数据显示格式和编辑操作，还提供了 Fill(填充)、Dump(输出)与 Polt(画图)等功能。

1) 改变存储器的数据显示格式

右击存储器显示窗口，从弹出的菜单中选择 Select Format，Select Format 子菜单中包含所有数据格式，或直接用 Ctrl+t 快捷键来转换数据显示格式。

2) 跳到某一地址上

右击存储器窗口，从弹出的菜单中选择 Go To，会打开一个 Go to Address 对话框(或直接利用快捷键 Ctrl+g 来打开 Go to Address 对话框)，在此对话框中输入十六进制的地址或通过 Browse 从标号列表中选择一个标号，最后点击 OK 退出此对话框。存储器窗口会显示指定地址处的存储器内容。

3) 显示存储器段

选择 Memory → Memory Map，打开存储器段显示窗口。如果没有程序加载，此时显示 DSP 中所有存储器段；如果程序加载，此时显示链接描述文件中分配的物理存储器段。

4) 填充或输出存储器内容

Fill 把数据文件(.dat)填充到存储器中，Dump 把存储器内容写到数据文件(.dat)中。通过如下步骤来 Fill 或 Dump 存储器内容：

右击存储器窗口，从弹出的菜单中选择 Fill 或 Dump(或选择 Memory → Fill 或 Dump)，打开一个 Fill Memory 或 Dump Memory 对话框，如图 4.8 所示。在此对话框中设置 Fill 或 Dump 操作，最后点击 OK，就启动了 Fill 或 Dump 操作。

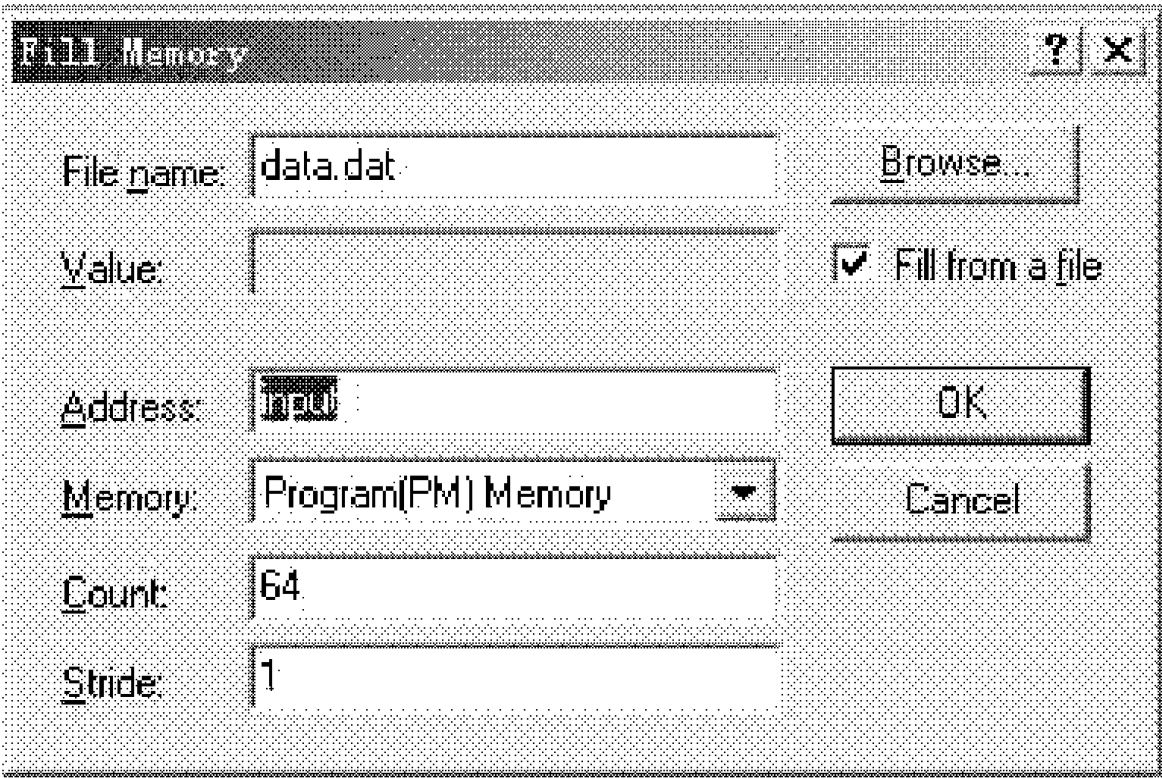


图 4.8　Fill Memory 对话框

(1) Fill 对话框描述：

File Name：指定填充数据文件的路径及文件名。

Fill From a File：选择此项，表示利用数据文件来填充存储器，否则用一个数值来填充。

Value：当 Fill From a File 没有选择时，在此项中输入一个十六进制数值，用此数值来填充存储器。

Address：填充存储器段的首地址。

Memory：选择存储器类型。

Count：填充的存储器单元数。

Stride：存储器单元步跳间隔，1 表示连续填充。

(2) Dump 对话框描述：

Address：输出的存储器段首地址。

File Name：指定输出数据文件的路径及文件名。

Memory：选择存储器类型。

Format：存储器内容的数据格式。

Count：输出的存储器单元数。

Stride：存储器单元步跳间隔，1 表示连续的存储器单元输出。

Write Format to File：当选择此项时，会把数据类型写到输出数据文件的开头。

5) 跟踪(Tracking)一个表达式

在存储器窗口中输入一个表达式来进行跟踪。

右击存储器显示窗口，从弹出的菜单中选择 New Tracking，打开一个 Enter A New Tracking Expression 对话框，在此对话框中输入一个表达式。此表达式可以是 C/C++表达式或寄存器表达式。如果是寄存器表达式，必须用$Xn 的形式：$表示寄存器，X 为寄存器名，n 为寄存器号。最后点击 OK，退出此对话框。存储器显示窗口的标题栏会显示此表达式。

6) 图形显示存储器内容

VisualDSP++的调试工具可以把一段存储器内容以图形方式画出来，并且它具有多种图形显示方式，包括一维 plot 图、二维 plot 图、星图和眼图等，而且在显示之前它还可以对数据进行处理，包括取对数(dB 显示)和 FFT 等。对图形的显示设置和操作也非常方便。

按下述步骤来对一段存储器内容进行图形显示：

(1) 选择 View→Debug Window→Plot→New，打开一个 Plot Configuration 对话框。如图 4.9 所示。

(2) 在 Plot Configuration 对话框中设置数据显示方式。

Type：设置数据显示类型，从其下拉菜单中选择：line plot、X－Y plot、Constellation plot(星图)、Eye Diagram(眼图)、Waterfall plot(瀑布图)或 Spectrogram plot(谱图)。

Title：输入图形窗口的标题名。

Name：在一个图形窗中可以同时画出多组数据，此项输入每组数据名。

Memory：指定存储器类型。

Address：存储器中数据块的首地址。

Offset：表示相对于首地址的偏移量，即以 Address+Offset 作为数据首地址进行显示。

Count：显示的存储器单元数。

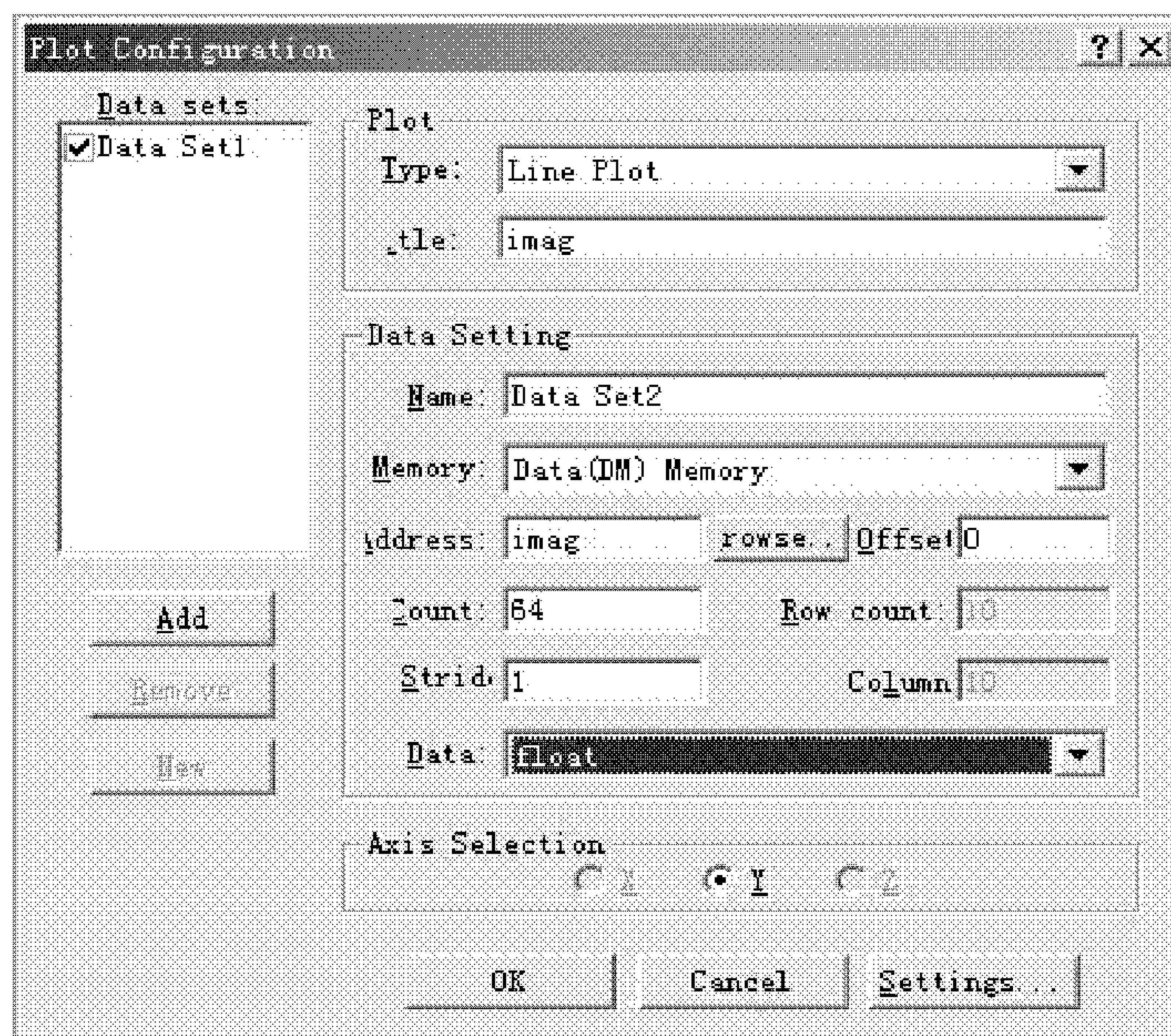


图 4.9　Plot Configuration 对话框

Stride：存储器单元步跳间隔，1 表示对连续存储器单元中的数据进行显示。

Data：从下拉菜单中选择存储器内容的数据类型。

Row count/Column：对于 Waterfall 和 Spectrogram 显示类型，这两项分别对行、列计数。

Axis Selection：分别指定数据的显示轴。

(3) 点击 Add，添加一个数据组，继续添加其它数据组，这样，在一个图形窗口中可以同时显示多组数据。

(4) 点击 Settings，进入 plot Settings 对话框，在 plot Settings 对话框中设置图形属性及对显示数据的处理。

(5) 点击 OK，退出 Plot Configuration 对话框，同时打开图形显示窗口，如图 4.10 所示。

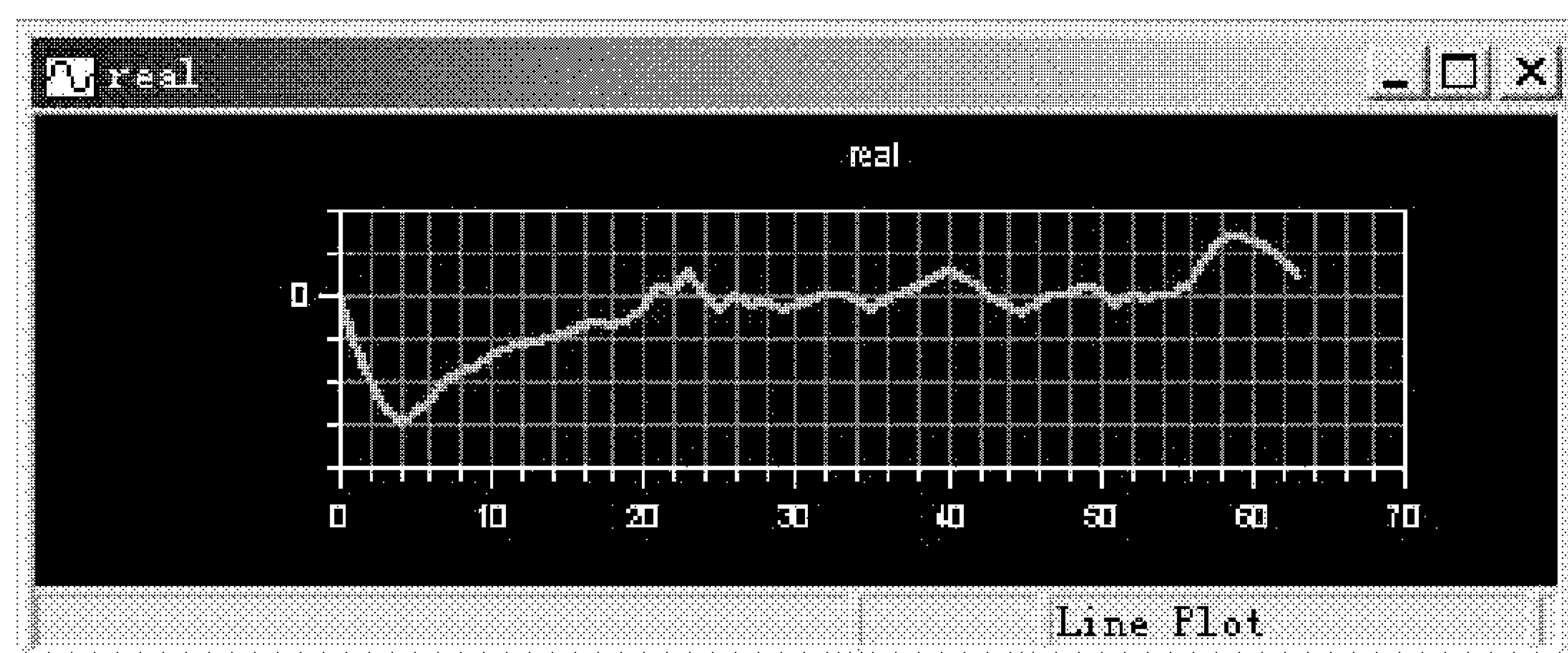


图 4.10　对存储器数据进行图形显示

之后，也可以对图形窗口重新配置：右击图形窗口，从弹出的菜单中选择 Configure，打开 Plot Configuration 对话框，对图形窗口重新配置。

8. 局部变量(Locals)观察窗口

局部变量(Locals)观察窗口中显示当前程序停止处所在的 C/C++函数体内的所有局部变量及其值。

选择 View→Debug Windows→Locals，打开 Locals 窗口。在此 Locals 窗口中会显示当前 C/C++函数内的所有局部变量及其值，可以修改这些局部变量的值及其显示格式。

9. 表达式(Expressions)值显示窗口

利用此窗口来显示当前表达式的值。此表达式可以是 C/C++表达式或寄存器表达式。如果是寄存器表达式，必须用$Xn 的形式：$表示寄存器，X 为寄存器名，n 为寄存器号。

选择 View→Debug Windows→Expressions，打开一个 Expressions 窗口，如图 4.11 所示。在 Expressions 窗口中允许写入一个表达式并显示其值，还可以改变值的显示格式。

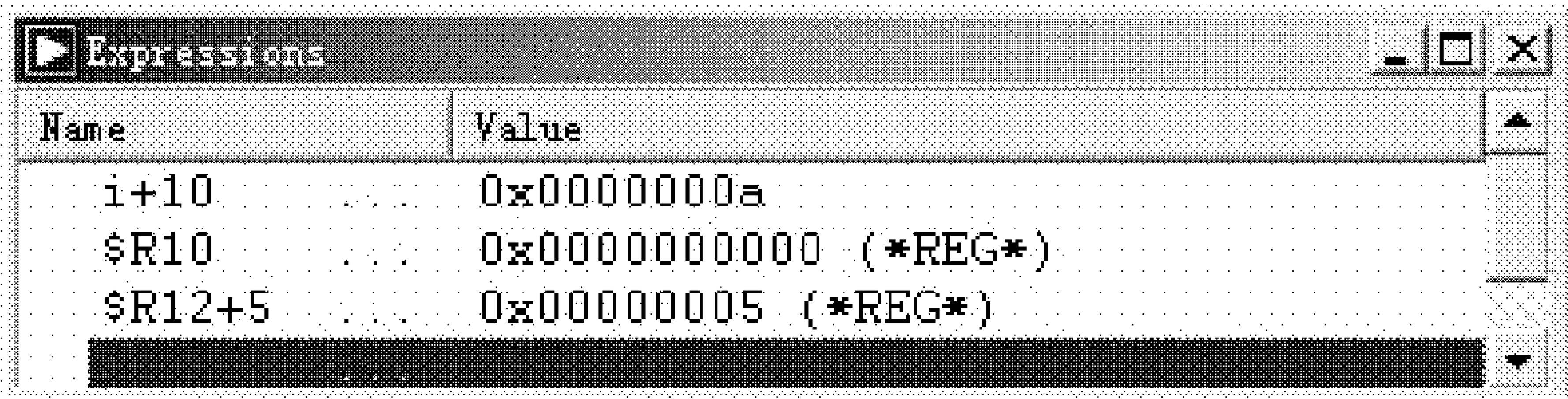


图 4.11　表达式值显示窗口

10. 堆栈观察窗口

C/C++编译器利用堆栈来保存暂时结果、函数调用时的参数传递、返回地址等，详见 2.3.7 节。利用此窗口来观察当前函数的压栈情况。

选择 View→Debug Windows→Call Stack，打开 Call Stack 窗。

11. 程序性能分析

VisualDSP++中提供了三种工具来分析程序的执行性能：Trace、Profile 和 Statistical Profiling。这三个工具都位于 Tools 菜单中，要显示这三种工具的执行结果还必须选择 View→Debug Windows→Trace 、Profile 或 Statistical Profiling Results，打开这些工具的执行结果窗口。

1) Trace 工具

Trace 提供对程序执行指令的跟踪，其结果显示了程序如何执行到某一地址上，即显示当前正在执行的指令、程序的取指(RD)、存储器数据的读(RD)、写(WR)等。通过如下步骤来设置 Trace 并显示其结果：

(1) 选择 Tools→Trace→Enable Trace，使能 Trace 工具。

(2) 选择 Tools→Trace→Set Trace Depth，打开 Trace Buffer Depth 对话框，设置用户指定的跟踪深度或最大跟踪深度。

(3) 选择 View→Debug Windows→Trace，打开 Trace 显示窗口，如图 4.12 所示。

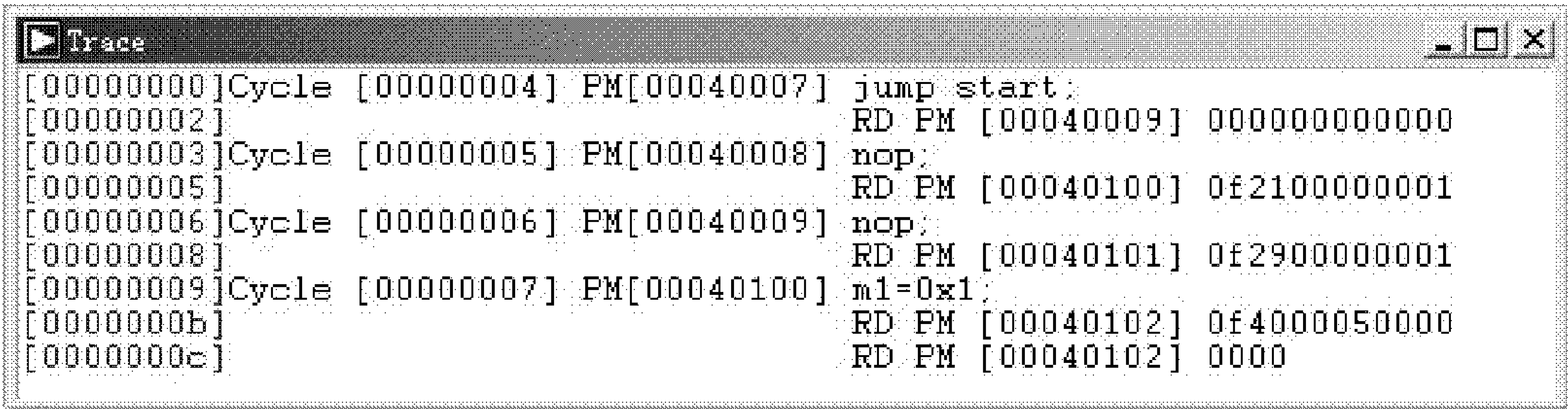


图 4.12　Trace 窗显示的结果

(4) 设置断点或单步运行程序，从 Trace 窗口中观察指令的执行。实际中需要与反汇编窗配合使用。

Trace 各部分描述如表 4.1 所示。

表 4.1　Trace 窗口描述

列	含　义
1	[xxxx]: 跟踪缓冲深度
2	Cycle[xxxx]: 指令从开始执行的周期数
3	PM[xxxx]: 执行指令的地址
4	反汇编指令，存储器结果具有如下定义：访问类型(RD 读或 WR 写)、存储器类型(PM 或 DM)、[地址]、读/写的数据值

2) Profile 工具

Profile 工具用来分析程序的运行时间特性，通过 Profile 可以找到最耗时的程序段，即需要进一步优化其性能的程序段。

通过下述步骤来设置 Profile 工具并显示其结果：

(1) 选择 Tools→Profile→Enable Profiling，使能 Profile 工具。

(2) 选择 Tools→Profile→Add/Remove Profile Ranges 工具，打开 Profile Ranges 对话框，如图 4.13 所示。

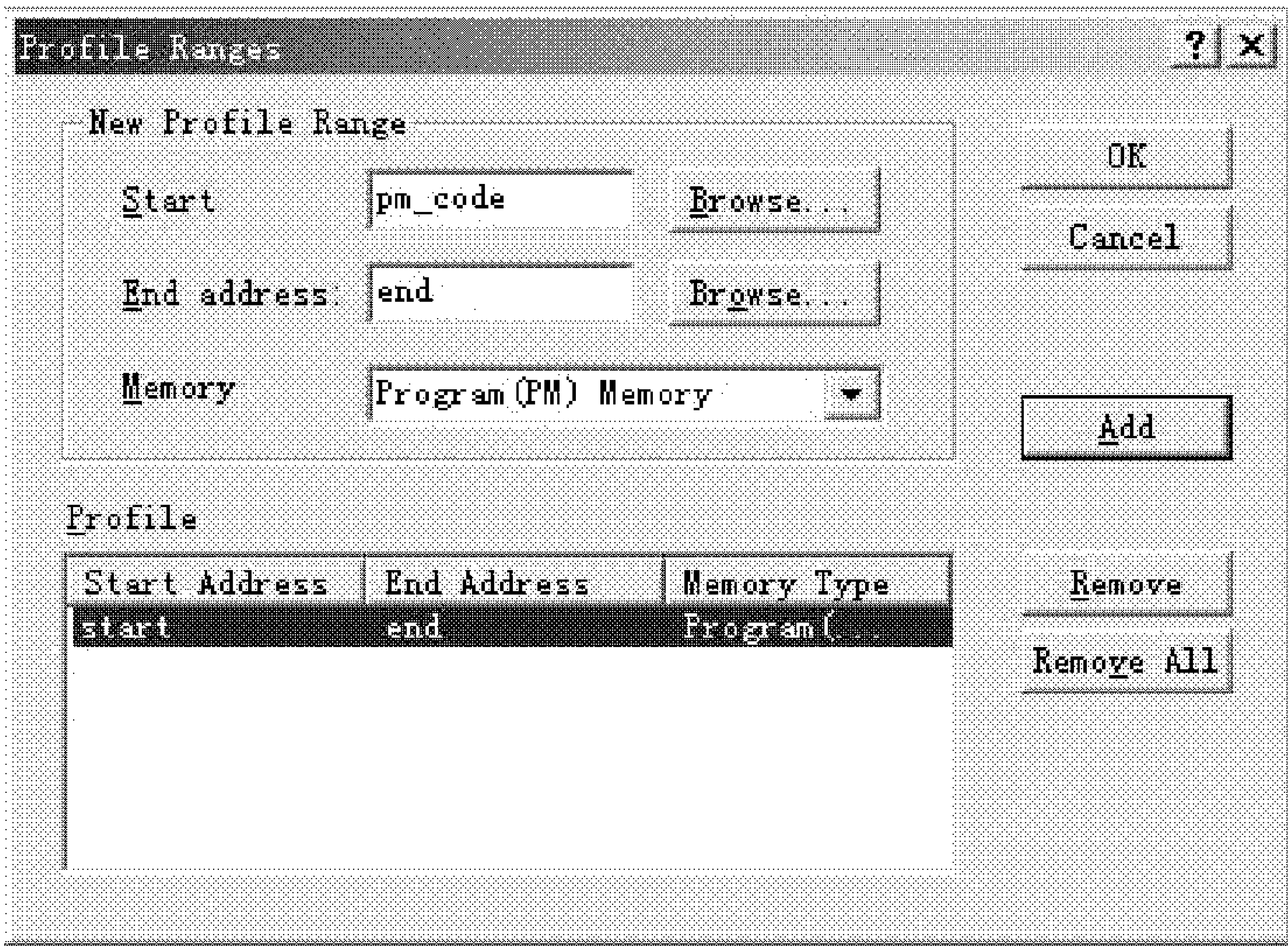


图 4.13　Profile Ranges 对话框

设置 Profile Ranges 对话框的参数：

Start Address：统计代码段的开始地址，可以是十六进制数或程序标号。

End Address：统计代码段的结束地址，可以是十六进制数或程序标号。

Memory Type：存储器类型。

Profile：列出所有激活的统计代码段。

Add，Remove，Remove All：添加、删除或删除所有的统计代码段。

(3) 设置好 Profile Range 后，点击 Add，所添加的统计代码段会出现在 Profile 的列表中，可以重复上述操作来添加多个统计代码段。

(4) 选择 View→Debug Windows→Profile，打开 Profile 窗口，如图 4.14 所示。

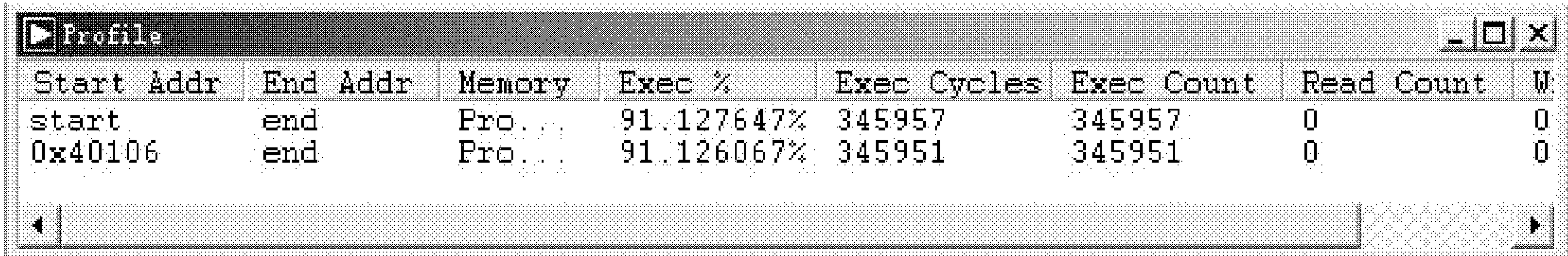


图 4.14　Profile 窗中显示的统计结果

Profile 窗口各部分的描述如表 4.2 所示。

表 4.2　Profile 窗描述

列	含　　义
Start Addr	代码段的开始地址
End Addr	代码段的结束地址
Memory	存储器类型
Exec%	此代码段的执行时间占整个程序执行时间的百分比
Exec Cycles	执行此代码段所需的总指令周期数
Exec Count	执行此代码段所需的总指令数
Read Count	执行此代码段时，所进行的存储器读的数目(包括指令取)
Write Count	执行此代码段时，所进行的存储器写的数目

3) Statistical Profiling 工具

类似于 Profile，Statistical Profiling 也是一个非常有用的工具，能够自动统计函数、源代码行的执行性能。

选择 Tools→Statistical Profiling→Enable Profiling，使能 Statistical Profiling 工具。每当重新加载可执行代码或运行时，VisualDSP++都会自动打开 Statistical Profiling Results 窗，也可以选择 View→Debug Windows→Statistical Profiling Results 来打开此窗。

当目标程序停止(碰到断点或手动停止)执行时，Statistical Profiling Results 窗中显示统计结果，如图 4.15 所示。

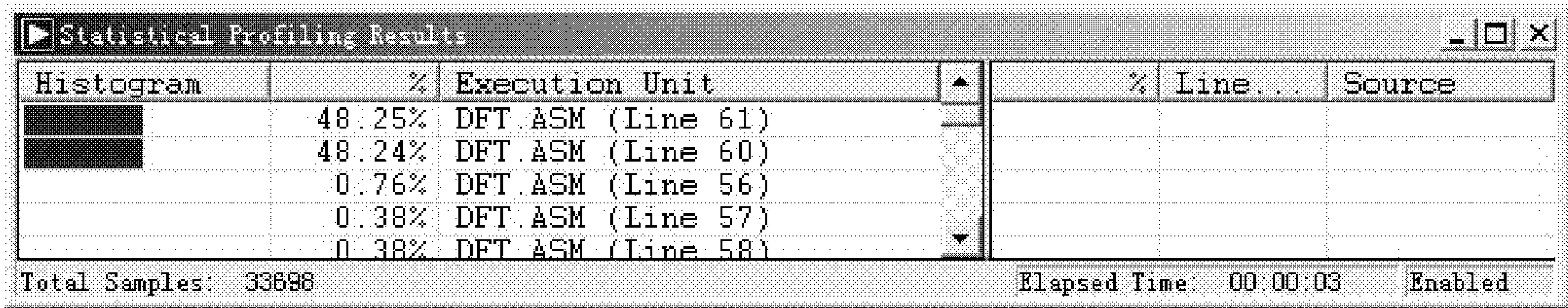


图 4.15　Statistical Profiling Results 窗的显示结果

Statistical Profiling Results 窗的左半部分为按执行时间大小排列的汇编代码行或 C/C++函数，右半部分显示源代码每一行的统计结果。统计结果既可以给出占用整个执行时间的百分比，也可以给出此行代码采样到的次数。

12．模拟硬件真实环境

VisualDSP++中提供了 3 个工具来模拟硬件环境：模拟中断(Interrupts)、模拟 I/O 数据流(Streams)和模拟 DSP 加载(Load Sim Loader)。Interrupts 模拟在程序的执行过程中产生的随机外部中断。Streams 模拟数据在 DSP I/O 管脚上的传送。Load Sim Loader 模拟通过 PROM、主机或链路对 DSP 进行加载。

1) 模拟中断

Interrupts 模拟在程序的执行过程中产生的随机外部中断，这对于调试中断服务程序是非常有用的。

通过下列步骤来模拟一个外部中断：

(1) 选择 Settings→Interrupts，打开 Interrupt Timing 对话框。

(2) 在 Interrupt Timing 对话框中设置中断，其各部分的描述如下：

External Interrupts：从下拉菜单中选择外部中断。

Min Cycles：中断信号产生的最小指令周期间隔。

Max Cycles：中断信号产生的最大指令周期间隔。

Offset Cycles：在第一个中断发生之前的指令周期数。

Interrupts：已配置的中断列表。

Add, Remove, Remove All：对中断列表进行操作。

(3) 点击 Add，把设置好的中断添加到中断列表中。

(4) 点击 OK，退出 Interrupt Timing 对话框并运行程序。

2) 模拟 I/O 数据流

Streams 模拟数据在 I/O 口的传送操作。

通过下列步骤来模拟一个数据传送：

(1) 选择 Settings→Streams，打开 Streams 对话框。

(2) 在 Streams 对话框中设置传送数据流，其各部分的描述如下：

Source/Destination：数据传送的源(设备)/目的(设备)。

Device：从下拉菜单中选择 I/O 设备。

Address：存储器映射 I/O 口的地址。

Mem Type：存储器映射 I/O 口的类型。

File/Browse ：选择一个数据文件。

Format：指定数据格式。

Circular：选择此项，则循环读此数据文件，即当读到文件尾时重新从文件头开始读。

Streams 对话框的 Active 栏中显示已设置的 I/O 数据流模拟。

(3) 点击 Connect。

(4) 点击 OK，退出 Streams 对话框。

(5) 运行程序。

3) 模拟 DSP 加载

Load Sim Loader 模拟通过 PROM、主机或链路给 DSP 加载.ldr 文件。

通过下列步骤来模拟 DSP PROM 加载：

(1) 选择 Settings→Load Sim Loader→Boot from PROM，打开 Open a Boot File 对话框。

(2) 选择指定的加载文件*.ldr 并打开。

(3) 选择 Debug→Reset。

(4) 点击工具栏中的 Run 图标，加载程序并停止。

(5) 设置断点并运行加载的程序。

模拟其它加载方式的步骤与上类似，这里不再一一介绍。

13．多 DSP 调试(多处理器共享总线)

很多实时处理的情况需要多片共享总线的 DSP 同时完成某项任务。要能正确的调试多片 DSP 系统，调试工具必须能够提供同步运行、同步单步执行、同步停止等，并能同时观察各 DSP 的程序执行情况。

在 VisualDSP 环境下，必须连接上硬件设备及其安装软件后才能进行多 DSP 调试。而 VisualDSP++提供了 Multiprocessor Simulator，可以模拟多 DSP 调试。

配置多 DSP 系统的第一步就是利用链接器的多 DSP 功能和链接描述文件(.LDF)来开发多 DSP 工程，第二步就是用 JTAG ICE Configurator 程序来描述与 VisualDSP++连接的硬件情况。如果是 Multiprocessor Simulator，则需要在 Simulator 的 New Session 对话框中配置多 DSP 系统。

描述完多 DSP 系统后，接下来就可以编译链接这个应用工程，并加载此工程。下面以 Multiprocessor Simulator 为例，介绍如何配置多 DSP 调试任务。

(1) 配置多 DSP 系统：选择 Session→New Session，打开 New Session 对话框。在 New Session 对话框中选择多 DSP 软件模拟器(MP Simulator)，如图 4.16 所示。

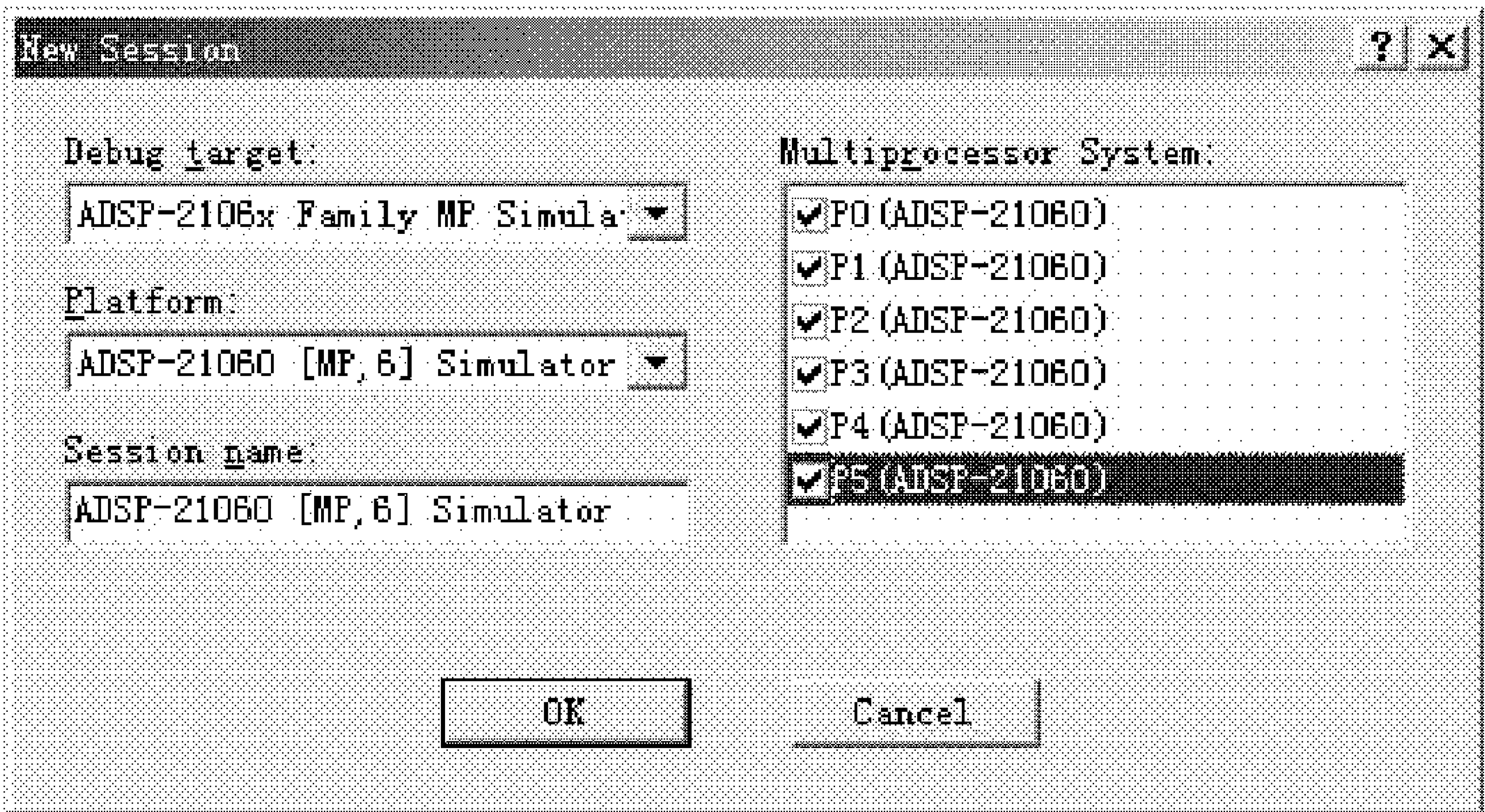


图 4.16 配置多处理系统

(2) 配置完多 DSP 系统后，点击 OK 退出 New Session 对话框。VisualDSP++会自动打开 P0 的反汇编窗和一个 Multiprocessor 窗，如图 4.17 所示。仔细观察 VisualDSP++主界面，可以看到此时 VisualDSP++的工具栏中多了 5 个多 DSP 同步程序控制命令：MP Run、MP Halt、MP Restart、MP Step 和 MP Reset，用来同时控制一组中的所有 DSP。

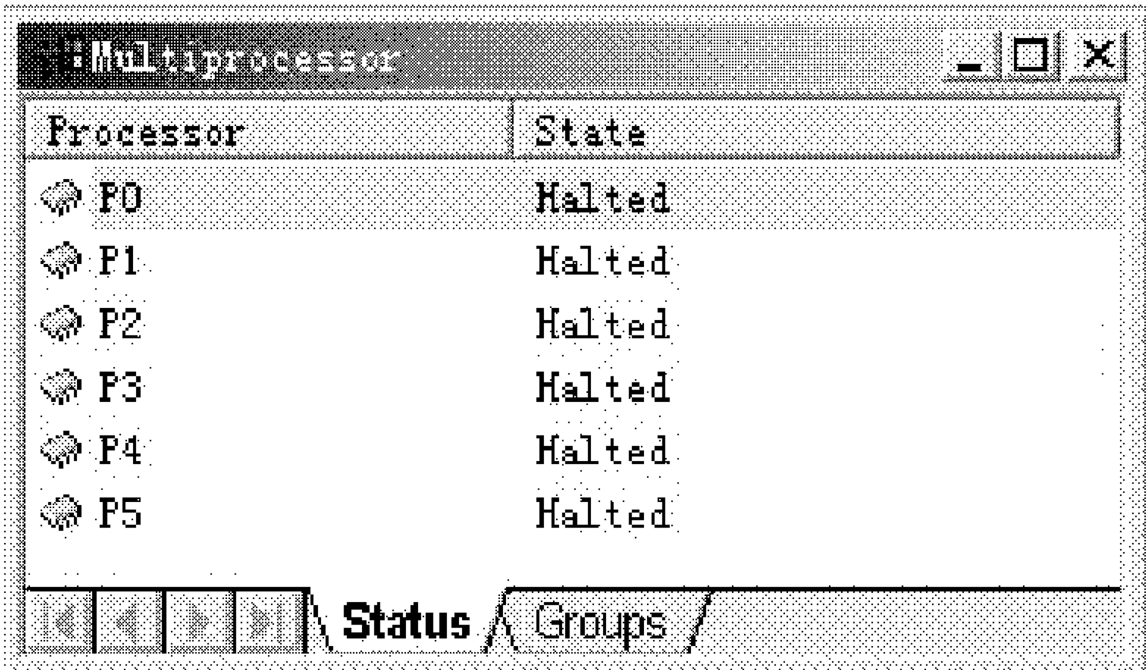


图 4.17 Multiprocessor 窗

有时需要几个 DSP 同步配合完成某一相关的任务，这几个 DSP 应该位于同一组中，以便利用 VisualDSP++的 MP 同步程序控制命令，同步控制组中所有 DSP 的运行。前面在配置多处理系统时，默认把所有的 DSP 都放入组 Default 中。用户也可以重新定义组，并把任意 DSP 分配到此组中：打开 Multiprocessor 窗的 Groups 栏，在 Groups 中右击，从弹出的菜单中选择 Add New Group，添加一个新组，并给此组分配 DSP；之后，首先选择组，然后才能利用 MP 同步命令来对此组中的所有 DSP 进行同步程序控制。

(3) 加载 MP 可执行程序：选择 File→Load Program，选择一个 MP 可执行文件，并打开此文件，同时会打开一个 Load Multiprocessor Confirmation 对话框。该对话框中列出了系统中的所有 DSP 及其已加载的可执行文件路径和文件名。在此对话框中为所有的 DSP 加载可执行代码。

(4) 确认所有 DSP 都加载了正确的可执行程序后，点击 OK，退出 Load Multiprocessor Confirmation 对话框。VisualDSP++的输出窗中显示所有 DSP 的加载状态。

在多 DSP 系统调试中，经常需要把寄存器、存储器和反汇编窗口等锁定到某一特定的 DSP 上，否则每当在 Multiprocessor 窗中选择另一 DSP 时，这些窗会随着变化。通过以下几步完成该功能：

(1) 在 Multiprocessor 窗中选择某一指定的 DSP。

(2) 打开需要查看此 DSP 的寄存器和存储器窗口等。

(3) 右击每个窗口，从弹出的菜单中选择 Pin to Processor。

多 DSP 调试工具与前面介绍的单 DSP 调试工具及其操作方法完全相同，只是 VisualDSP++为多 DSP 调试提供了同步程序控制功能。

4.4　VisualDSP++演示例子

下面通过一个简单的例子来演示 VisualDSP++ 环境下的程序开发过程以及如何利用 MATLAB 产生数据文件并对 DSP 处理结果进行分析。这个例子的源文件在安装目录下的 VisualDSP\211xx\Examples\ASM_Examples\Dft\目录中，把此目录中的源文件 dft.asm 和 dft.ldf 复制到用户目录：VisualDSP 安装目录\211xx\Examples\ASM_Examples\myproject\Dft\中。

按下列步骤完成创建工程、编译链接生成可执行文件、加载运行并调试的整个过程。

(1) 在 WINDOWS 的开始菜单中选择 Programs → VisualDSP → VisualDSP++ for SHARC，打开 VisualDSP++主界面。VisualDSP++会自动把上次运行时的工程内容打开，因此应首先关闭这些上次运行时打开的窗口，并选择不要保存。

(2) 选择 Session→New Session，打开 New Session 对话框，在此对话框中指定目标调试平台：

Debug targer: ADSP－2116x Family Simulator

Platform: ADSP－2116x Simulator

Processor: ADSP－21160

(3) 选择 Project→New，打开一个 Save New Project As 对话框，在此对话框中输入工程名 dft，并选择保存路径 VisualDSP\211xx\Examples\ASM_Examples\myproject\Dft\，保存此新工程。

(4) 退出保存对话框时，会自动弹出 Project Options 对话框，在此对话框内设置工程选项：

Processor: ADSP－21160　　Type: DSP executable file

Name: dft　　Settings for: Debug

(5) 完成设置 Project Options 对话框后，点击 OK 退出此对话框，同时会弹出一个询问是否在新工程中加入 VDK 的对话框，点击 No，退出此对话框。VisualDSP++界面上会出现一个标题为 Project: dft.pjt 的工程视窗，当前工程中没有添加任何文件，因此该视窗只显示了几个空文件夹。

(6) 右击工程视窗中的工程名，从弹出的菜单中选择 Add File(s) to Project，给此工程加入源文件 dft.asm 和 dft.ldf，加入的源文件会显示在相应的文件夹中。

(7) 双击工程视窗中的 dft.asm 文件，在编辑窗口中打开此源文件并查看其源代码，可以看到 dft.asm 还需要两个数据文件：test64.dat 和 sin64.dat，test64.dat 是输入 DFT 的测试数据，而 sin64.dat 是计算 DFT 时需要的旋转因子。产生数据文件最方便的方法就是利用 MATLAB 来产生。

(8) 利用 MATLAB 来产生数据文件。

MATLAB 具有强大的数据产生工具、数据处理工具、滤波器设计工具和数据可视化工具等，MATLAB 能够产生各种数据及数据文件，因此利用 MATLAB 来产生 DSP 所需的数据文件是最简单、方便的方法。

利用下面的一段 MATLAB 程序来产生 sin64.dat 和 test64.dat 数据文件：

```
sine=sin([0:63] *2*pi/64);          %生成正弦数据
save sin64.dat   sine -ascii        %输出到数据文件 sin64.dat 中
testdat=zeros(64,1);                %产生测试数据
testdat(1:4,1)=1.0;
save test64.dat   testdat -ascii    %输出测试数据到 test64.dat 文件中
```

把生成的数据文件 test64.dat 和 sin64.dat 复制到 VisualDSP++\211xx\Examples\ASM_Examples\myproject\Dft\下。

(9) 继续向工程 dft.pjt 中添加 test64.dat 和 sin64.dat 数据文件。

(10) 接下来就可以编译链接此工程：选择 Project→Build Projec 或点击工具栏中的 Build Project 图标。编译链接过程中，输出窗中会显示编译链接信息。如果出错，通过双击出错信息会自动打开出错的源文件。

(11) 编译链接成功后，加载生成的可执行代码 dft.dxe：选择 File→Load Program，选择 VisualDSP++\211xx\Examples\ASM_Examples\myproject\Dft\Debug\目录下的 dft.dxe，并打开。

如果工程配置中选择了 Debug，则默认生成的可执行文件在...\Debug\目录下。用户也可以在 Project Options 对话框的 General 面板栏中重新指定。

(12) 利用 VisualDSP++中的调试工具进行调试。

(13) 在 dft.asm 源文件的 end 标号处设置一断点，然后运行程序。

(14) 选择 View → Debug Windows → Plot → New，打开 Plot Configuration 对话框。在此对话框中设置画出 DFT 处理结果的实部(real)和虚部(imag)。关于 Plot Configuration 对话框的设置方法我们在前面已作了详细介绍，这里不再重复。VisualDSP++的图形显示结果如图 4.18 所示。

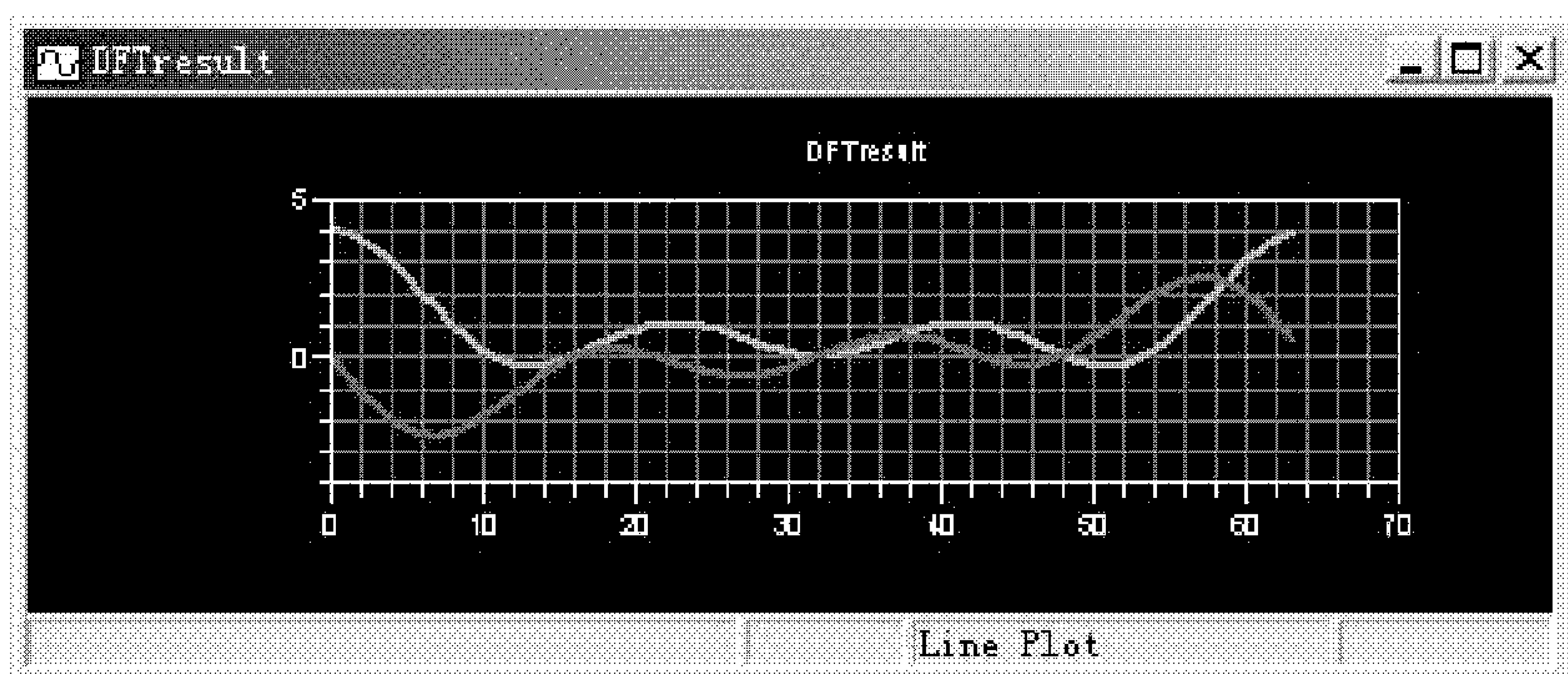


图 4.18　VisualDSP++的 DFT 处理结果的图形显示

(15) 为了验证 DSP 处理结果的正确性，需要利用 MATLAB 来进行同样的处理，并比较它们的结果。下面利用 MATLAB 对测试数据进行同样的处理，并画出其处理结果，如图 4.19 所示。

```
a=fft(testdat);
figure;    plot(real(a));    hold on;    plot(imag(a));
```

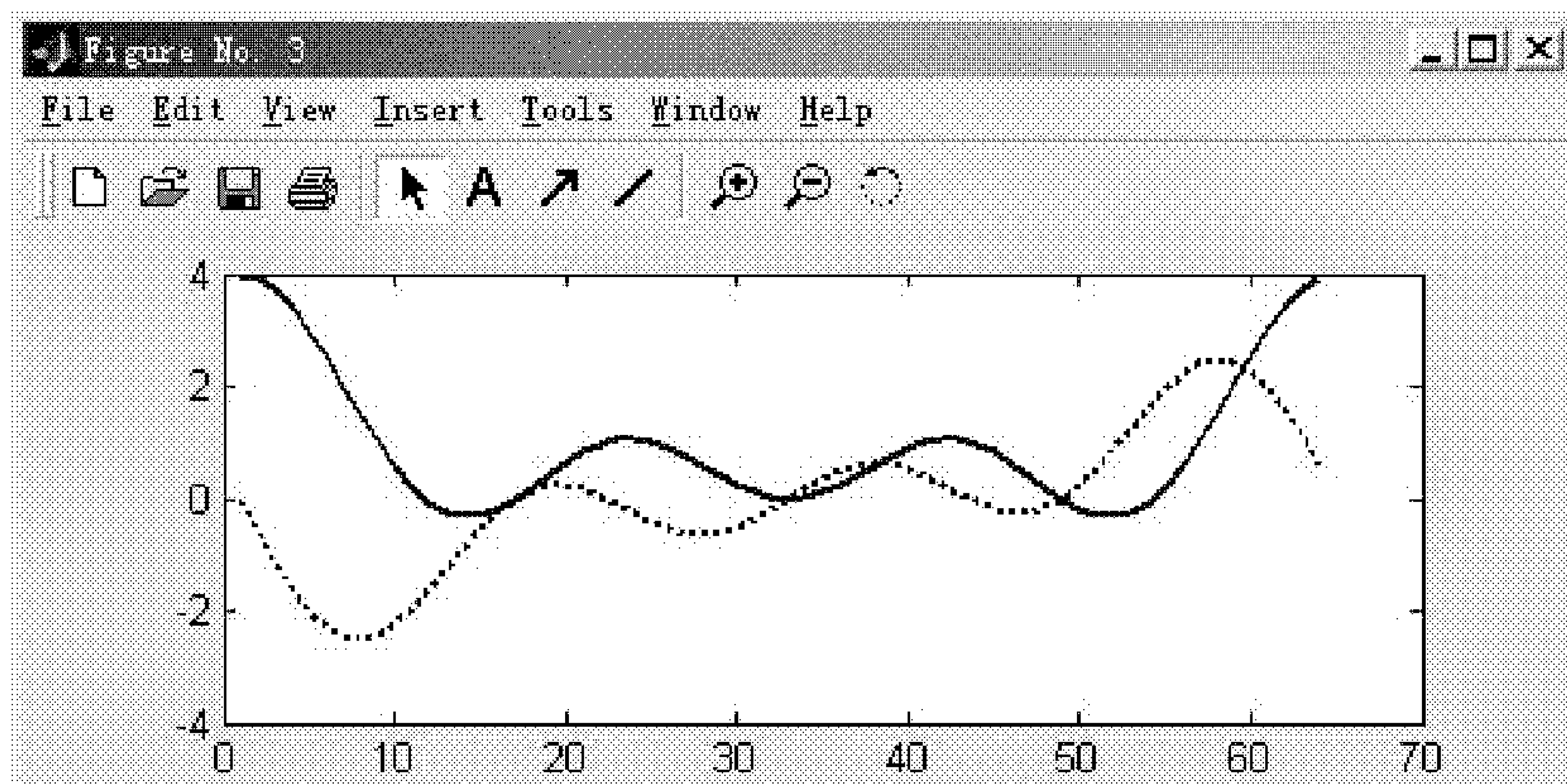


图 4.19　MATLAB 的 DFT 处理结果

比较图 4.19 和图 4.18，可以看出 DSP 处理结果与 MATLAB 的处理结果完全相同，这表明汇编程序 DFT 是正确的。

(16) 有时为了进行更精细的比较，需要把 DSP 处理的中间结果或最终结果输出到某一文件中，然后把此数据文件读入到 MATLAB 空间中，与 MATLAB 处理的中间结果或最终结果进行比较，从而可以快速查出 DSP 程序中存在的问题，因此 MATLAB 对于帮助 DSP 程序调试是非常有用的。下面演示如何把 DSP 的处理结果输出到某一数据文件中，并把此数据文件读入到 MATLAB 空间中进行分析。

选择 Memory→Dump，打开 Dump Memory 对话框，设置此对话框，把实部结果输出到数据文件 dft_real.dat 中(注意不要选择此对话框中的 Write format to file 项)。点击 OK，退出 Dump Memory 对话框，同时把以 real 为首地址的一段存储器内容输出到数据文件中。

利用 MATLAB 程序把 dft_real.dat 数据文件读入到 MATLAB 空间中，并把 DSP 处理结果和 MATLAB 处理结果画在一起，如图 4.20 所示。

```
load dft_real.dat -ascii          %把 real.dat 读入到 MATLAB 空间中
figure;   plot(dft_real, '*r'); hold on; plot(real(fft(testdat)), 'k');   %把两者结果画在一起比较
```

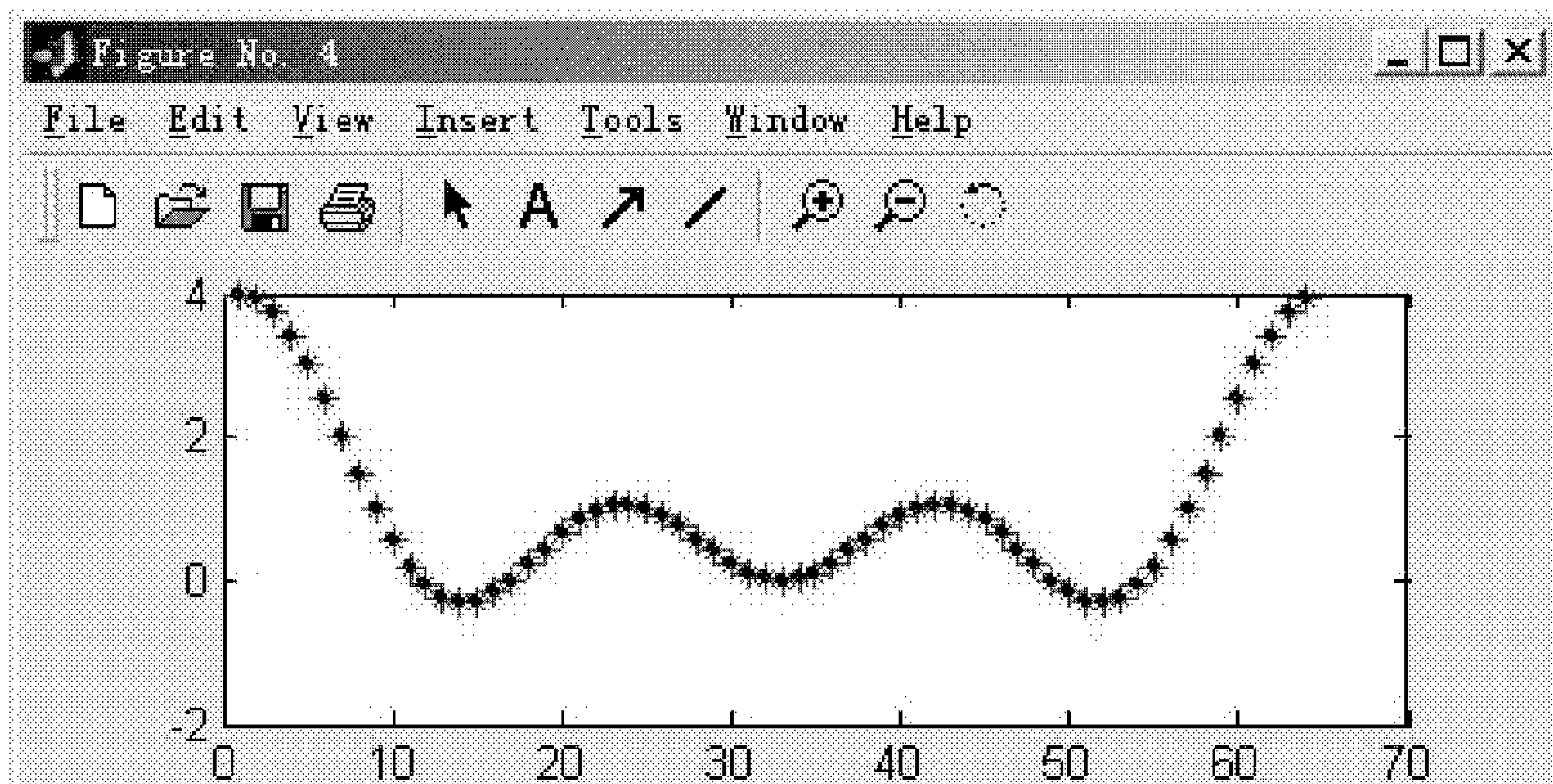


图 4.20　MATLAB 与 DSP 处理结果比较

从图 4.20 可以看出，两者处理结果完全相同。

另外，类似于第 5 章和第 6 章介绍的 MATLAB 与 TI CCS 的接口，SDL 开发商也为 SHARC DSP 开发了一套工具：DSPdeveloper for SHARC，把此工具和 MATLAB 结合，可以直接把 Simulink 模型生成 SHARC DSP 的可执行代码，在统一环境下完成设计、仿真、代码生成、调试和运行。本书限于篇幅，仅在第 7 章对此做了简要介绍，感兴趣的读者可以登录到 www.sdltd.com/dspdeveloper，能够得到更多详细信息。

思　考　题

4.1　VisualDSP++是 AD 公司推出的 DSP 开发环境，它可以支持 AD 公司的哪些类型的 DSP？VisualDSP++3.0 提供了哪些功能？VisualDSP++3.0 与第 3 章介绍的 CCS2.0 在功能、界面、操作方法等方面有何相同点？

4.2　用 C/C++语言编写一个 FFT 程序，练习应用 VisualDSP++中提供的编辑工具、代码生成工具和调试工具，最终生成 SHARC DSP 的可执行代码。

4.3　VisualDSP++3.0 中的文件类型有哪些？每种文件类型的后缀和功用是什么？这些文件类型之间的关系是什么？

4.4　什么是链接描述文件？有什么用途？利用 C/C++语言混合编程和只利用汇编语言编程时的链接描述文件是否相同？如何修改？VisualDSP++3.0 中的链接描述文件(.ldf)与 CCS 中的链接命令文件(.cmd)的功能是否相同？

4.5　VisualDSP++3.0 提供的调试工具有哪些？监视点(watchpoint)和断点(breakpoint)的区别和相同点是什么？VisualDSP++3.0 中是否也提供了探点(probepoint)功能？

4.6　VisualDSP++ Kernel (VDK)有什么用途？类似于 CCS 中的 DSP/BIOS，VDK 也是一种实时操作系统，可以完成何种功能，如何应用？

4.7　VisualDSP++3.0 中的 VCSE(Visual Component Software Engineering)有什么功能？如何应用？

4.8　VisualDSP++3.0 也是一种开放式结构，类似于 MATLAB Link for CCS Development Tools，用户是否也可以利用 MATLAB 来直接访问 VisualDSP++3.0 中的目标数据？

4.9　如何利用 MATLAB 来产生目标 DSP 程序的测试数据？如何把 MATLAB 产生的测试数据输入到目标 DSP 中？反之又如何呢？

第 5 章　MATLAB 与 TI CCS 的接口

众所周知，MATLAB 具有强大的分析、计算和可视化功能，使用非常方便。而且我们在开发新的数字信号处理算法时，总是先用Matlab 进行仿真，当仿真结果满意时再把算法修改成 C/C++(或汇编)语言，在硬件的 DSP 目标板上实现。编写 C/C++(或汇编)语言算法与编写Matlab 算法当然不是一个概念，前者复杂得多，因此我们经常做的工作就是通过开发工具 CCS(或 AD 公司的 DSP 开发工具 VisualDSP++)把目标 DSP 程序运行的中间结果保存到 PC 机的硬盘上，然后再调入到 MATLAB 工作空间中，与 MATLAB 仿真算法的中间结果进行比较，以发现 DSP 程序中由设计或精度问题导致的结果偏差，如此过程反复进行，非常不方便。对此，开发人员梦寐以求一种新的工具，能够把 MATLAB 和 DSP 开发工具集成在一起。

MathWorks 公司和 TI 公司联合开发的 MATLAB Link for CCS Development Tools，提供了 MATLAB 和 CCS 的接口(本书称之为连接)，即把 MATLAB 和 TI CCS 及目标 DSP 连接起来。MATLAB Link for CCS Development Tools 作为 MATLAB 的一个新工具箱被集成在 MATLAB 6.5(Release13)中。利用此工具可以像操作 MATLAB 变量一样来操作 TI DSP 的存储器或寄存器，即整个目标 DSP 对于 MATLAB 似乎是透明的，开发人员在 MATLAB 环境下就可以完成对 CCS 的操作。例如，调用 DSP 目标程序中的函数，读写 DSP 中的某一段存储器或寄存器，利用 RTDX 实时数据交换等，所有这一切操作只需利用 MATLAB 命令和对象就能实现，简单、方便、快捷。MATLAB Link for CCS Development Tools 可以支持 CCS 能够识别的任何目标板，包括 TI 公司的 DSK、EVM 板和用户自己开发的目标 DSP(C2000™，C5000™，C6000™)板。

本章详细介绍 MATLAB Link for CCS Development Tools 的功能、所提供的命令、对象以及它们的使用方法等内容。通过本章的学习，读者就可以感受到 MATLAB Link for CCS Development Tools 所带来的简单、方便、快捷，并且会使读者更增加开发嵌入式实时应用系统的乐趣。

为了表述简捷，本章把 MATLAB Link for CCS Development Tools 简称为 CCSLink。

5.1　CCSLink 概述

5.1.1　CCSLink 的功能及特点

集成在 MATLAB 6.5 中的 CCSLink 工具把 MATLAB、TI 开发环境(CCS)及硬件 DSP 连

接起来，允许开发者在 MATLAB 的环境下就可以完成对 CCS 和硬件目标 DSP 的操作，它提供了 MATLAB、CCS 和目标 DSP 的双向连接，开发者可以利用 MATLAB 中强大的可视化、数据处理和分析函数对来自 CCS 和 TI DSP(C2000™，C5000™，C6000™ 系列)的数据进行分析和处理，这样大大简化了 TI DSP 软件开发的分析、调试和验证过程。

利用 CCSLink 工具，可以把数据从 CCS 中传送到 MATLAB 中去，也可以把 MATLAB 中的数据传到 CCS 中。而且通过 RTDX(实时数据交换)技术，可以在 MATLAB 和实时运行的 DSP 硬件之间建立连接，在它们之间实时传递数据而不使 DSP 正在运行的程序停止。这种能力给开发者提供了一个观察 DSP 实时运行情况的“窗口”，开发者可以在 MATLAB 中改变一个参数或变量，并把此值传递给正在运行的 DSP，从而可以实时地调整或改变处理算法。如果 CCSlink 和 Embedded Target for TI C6000 DSP Platform(集成在 MATLAB 6.5 中的另一种专门为 TI C6000 DSP 开发的产品，我们将在第 6 章中介绍)相结合，可以直接由 Simulink 模型生成 TI C6000 DSP 的可执行代码，开发者可以在统一的 MATLAB 环境中完成原理设计、仿真、调试、测试和在 TI C6000 DSP 上运行。MATLAB、CCSLink、CCS 和硬件目标 DSP 的关系如图 5.1 所示。

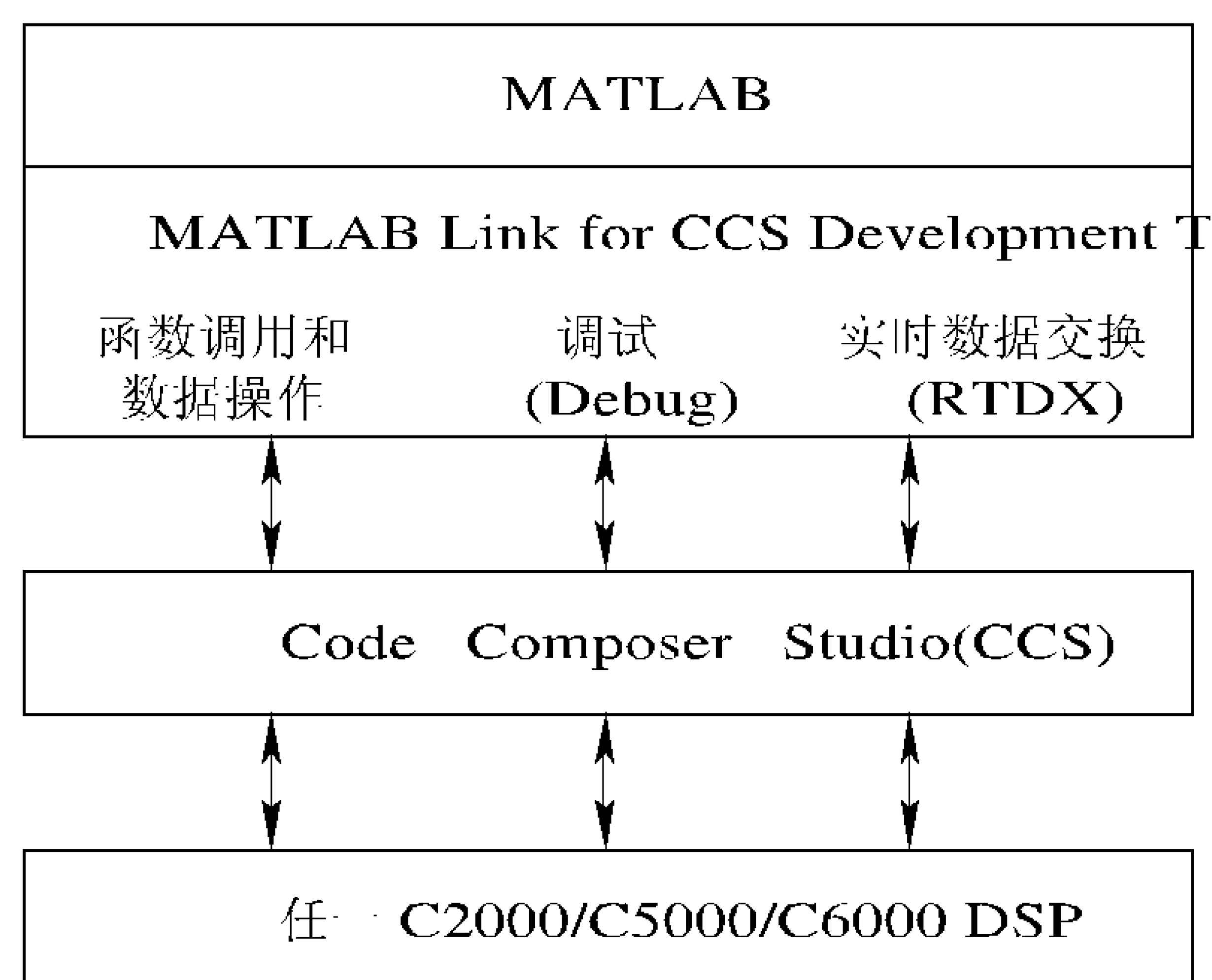


图 5.1 CCSLink 把 MATLAB 和 TI 开发工具及目标 DSP 连接在一起

CCSLink 主要特点总结如下：

- 提供 MATLAB 函数，可以自动完成调试、数据传递和验证。
- 在 MATLAB 和 DSP 之间实时传递数据，而不用停止 DSP 中程序的执行。
- 支持 XDS510/XDS560 仿真器，可以高速调试硬件 DSP 目标板。
- 提供嵌入式对象，可以访问 C/C++变量或数据。
- 对测试、验证和可视化 DSP 代码提供帮助。
- 扩展了 MATLAB 和 eXpressDSP 工具的调试能力。
- 符合 TI eXpressDSP™ 标准。

5.1.2 CCSLink 的配置

MATLAB 6.5(R13)或更新版本集成了 CCSLink 工具，当前的 CCSLink 版本为 1.0。CCSLink 可以支持 CCS 能够识别的任何板卡及其硬件 DSP，包括 TI C2000™、C5000™、C6000™ DSP 及 EVM(评估板)、DSK(初学套件)、simulator(软件模拟器)以及任何符合标准的用户板和第三方板卡。

包括上述硬件，CCSLink 还需要 Mathworks 公司和 TI 公司的软件产品支持，CCSLink 需要的软件产品包括：

Mathworks 公司：MATLAB 6.1(或更新版本)，信号处理工具箱 5.0(或更新版本)。

TI 公司：编译器(compiler)、汇编器(assembler)、链接器(linker)、CCS IDE 2.1、CCS 配

置工具及其它工具。

验证 CCSLink 是否在主机系统上安装成功，要在 MATLAB 命令窗中输入命令：

```
help ccslink
```

如果 CCSLink 安装成功了，则 MATLAB 命令窗中会显示如下产品信息：

```
MATLAB Link for Code Composer Studio(tm)
Version 1.0 (R13) 28-Jun-2002


Methods for MATLAB Link for Code Composer Studio
ccshelp/ccsdsp          - Construct CCS object.
```

如果 MATLAB 不能返回任何信息，就需要重新安装 CCSLink。

验证 CCS 是否也在主机系统上安装并配置好，要在 MATLAB 命令窗中输入如下命令：

```
ccsboardinfo
```

如果 CCS 已经安装并配置好，则 MATLAB 命令窗中会返回类似如下的板卡信息：

```
Board   Board                                   proc    processor       processo
Num     Name                                    Num     Name            Type
……      ……………………………………………………              ……      ……………………………………………………
1       C6xxx   Simulator (Texas Instrum …      0       6701            TMS320C6701
0       C6x11   DSK (Texas Instruments)         0       CPU             TMS320C6x1x
```

如果 MATLAB 不能返回任何板卡信息，就需要重新安装并配置 CCS。

最后还要确定 CCS 是否能够正确启动，因为 CCSLink 工作时会自动启动 CCS。

5.1.3　CCSLink 的内容

开发者利用 CCSLink 提供的 MATLAB 函数完成 MATLAB 与 CCS 和目标 DSP 的存储器及寄存器中的信息之间的交换。CCSLink 提供了三个组件内容：

(1) 与 CCS IDE 的连接对象。利用此对象来创建 CCS IDE 和 MATLAB 的连接。从 MATLAB 的命令窗中可以运行 CCS IDE 中的应用程序，向硬件 DSP 的存储器或寄存器发送或取出数据，检查 DSP 的状态，而且可以开始和停止 DSP 上运行的程序。

(2) 与 RTDX 的连接对象。提供 MATLAB 和硬件 DSP 之间的实时通信通道。利用此连接对象，可以打开、使能、关闭或禁止 DSP 的 RTDX 通道，利用此通道可以实时地向硬件目标 DSP 发送和取出数据而不用停止 DSP 上正在执行的程序。例如把原始数据发送给程序进行处理，并把处理结果取回到 MATLAB 空间中进行分析。

RTDX 连接对象实际上是 CCS 连接对象的一个子类，在创建 CCS 连接对象的同时创建 RTDX 连接对象，它们不能分别创建。

(3) 嵌入式对象。在 MATLAB 环境中创建一个可以代表嵌入在目标 C 程序中的变量的对象。利用嵌入式对象可直接访问嵌入在目标 DSP 的存储器和寄存器中的变量，即把目标 C 程序中的变量作为 MATLAB 的一个变量对待。在 MATLAB 中收集 DSP 程序中的信息，转变数据类型，创建函数声明，改变变量值，并把信息返回到 DSP 程序中，所有这些操作都在 MATLAB 环境下完成。

实际上，无论连接对象还是嵌入式对象都作为 MATLAB 中的一种对象来对待。与 MATLAB 中的所有面向对象编程一样，可以设置和获取对象的属性及其属性值，即在 MATLAB 中对所有对象的操作方式是一样的，只是对应的属性及属性值不同而已。

嵌入式对象利用连接对象来访问目标 DSP 的存储器和寄存器内容，因此在利用嵌入式对象之前必须先创建连接对象。

在本章的下面几节中将详细介绍这些对象、对象属性及其操作函数。

5.2　CCSLink 的连接对象

CCSLink 利用 MATLAB 的面向对象编程技术创建了两种连接对象：CCS IDE 连接对象和 RTDX 连接对象。RTDX 连接对象实际上是 CCS IDE 连接对象的一个子类。

在这一节中我们讨论这两种对象的创建方法、对象的属性和属性值以及如何设置和获取属性等。

5.2.1　创建连接对象

创建连接对象的最简单方法是利用函数 ccsdsp 来创建一个具有默认属性值的连接对象。例如创建一个 CCS IDE 连接对象时，在 MATLAB 命令窗中需输入如下命令：

```
cc=ccsdsp
```

cc 为 CCS IDE 连接对象的句柄。如果上述 MATLAB 命令后面没有加分号，MATLAB 命令窗中会列出连接对象 cc 的属性及其默认的属性值：

```
CCSDSP   object:
  API       version         : 1.0
  Processor   type          : C67
  Processor   name          : CPU
  Running?                  : No
  Board number              : 0
  Processor number          : 0
  Default timeout           : 10.00 secs
  RTDX channels             : 0
```

观察上面 MATLAB 命令窗中列出的属性信息，可以看到 CCS IDE 连接对象和 RTDX 连接对象是同时创建的，它们不能分别创建，它们保持一个类成员关系。RTDX 连接对象是 CCS 连接对象的一个成员，如果输入如下命令：

```
rx=cc.rtdx
```

则 rx 成为 CCS IDE 对象中 RTDX 成员的一个句柄。在对 RTDX 通信通道进行操作的函数中(例如 readmat 和 writemsg)，rx 可以替代 cc.rtdx 作为这些函数的输入参数。在 MATLAB 命令窗中输入 rx，会显示如下 rx 属性信息：

```
rx
RTDX channels           : 0
```

5.2.2 修改和获取连接对象的属性值

类似于 MATLAB 的其它对象，CCSLink 中的对象也具有一些预先定义的域，称为对象属性，而且每一属性都分配一个值，叫做属性值。属性值有些可以设置，有的则不能。用户可以在创建对象时设置属性值，也可以在创建之后再改变这些属性值。有些属性值是只读的，因此不能设置。而有的属性值，例如目标板号和 DSP 号，在创建对象设置完成后就变成只读的了，因此在创建之后就不能再修改它们。

1. 创建连接对象时直接设置属性值

通过在 ccsdsp(连接对象创建函数)中添加入口参数，可以在创建连接对象的同时直接设置属性值，应注意：

- 表示属性名的字符串放入单引号内，并且属性名之间用逗号相隔；
- 每一属性名的后面是其设置的属性值，也以逗号相隔，如果属性值也为一字符串，也要将其放入单引号内。

例　cc=ccsdsp('boardname', 1, 'procnum', 0, 'timeout', 5);

boardname, procnum, timeout 都是属性名(我们在后面再详细介绍)。上例中的 MATLAB 命令表示：创建一个连接对象，此连接对象与主机上的第二个 DSP 板(0 表示第一个)和此板上的第一个 DSP 建立连接关系，并设置全局超时值为 5 s(默认为 10 s)。

利用此方法可以同时对多个属性进行设置。

2. 利用 set 函数设置属性值

当创建完一个连接对象后，可以利用 set 函数来修改此连接对象的属性值。

例　set(cc, 'timeout', 8);

利用 set 函数修改完属性值后，可以再利用 get 函数来查看修改后的属性值。

例　get(cc)

```
ans=
        rtdx: [1x1 rtdx]
        apiversion: [1 0]
        ccsappexe: [ ]
        boardnum: 0
        procnum: 0
        timeout: 8
        page：0
```

3. 利用 get 函数获取对象属性

例　v=get(cc, 'apiversion')

```
ans=
 1      0
```

上例用 get 函数获取连接对象句柄 cc 的 apiversion 属性值，并把此值分配给一个变量 v。

4. 利用直接属性查询方法来设置和获取属性值

利用类似于 MATLAB 结构体查询的方法来设置和获取一个对象的属性值。

例
```
cc=ccsdsp;
cc.time=6;
cc.rtdx.numchannels=4;
```
上例首先创建一个具有默认属性值的连接对象，然后利用直接属性查询方法来修改超时和RTDX 通道数目的属性值。可以利用同样的方法来获取属性值。

例
```
num=cc.rtdx.numchannels
    num=
        4
```
上例利用直接属性查询方法来获取 RTDX 通道数目的属性值，并把此值分配给 num 变量。

5.2.3　连接对象属性

CCSLink 提供了 MATLAB 与 CCS 和目标硬件的连接，开发人员利用此连接使 MATLAB 与 CCS 和目标 DSP 进行通信。每一连接包括两个对象：CCS IDE 连接对象和 RTDX 连接对象。每一对象都具有多个属性，通过设置这些属性值来配置对象。

本节详细介绍 CCS IDE 和 RTDX 连接对象的属性，包括属性名、使用方法、是否可以设置和获取属性值等。表 5.1 首先列出 CCS IDE 和 RTDX 连接对象的所有属性，接下来再对所有属性进行详细介绍。

表 5.1　连接对象属性

属性名	应用哪一对象	用户是否可设置	描　述
apiversion	CCS IDE	否	报告 CCS IDE API 的版本号
boardnum	CCS IDE	是/创建时	指定 CCS IDE 可识别目标板的索引号
ccsappexe	CCS IDE	否	指定 CCS IDE 可执行文件所在的路径
numchannels	RTDX	否	某一连接对象所打开的 RTDX 通道数
page	CCS IDE	是/默认	默认的存储器页
procnum	CCS IDE	是/创建时	目标板上 DSP 的索引号
rtdx	RTDX	否	指定 RTDX
rtdxchannel	RTDX	否	标识 RTDX 通道的字符串
timeout	CCS IDE	是/默认	连接对象的全局超时设定
version	RTDX	否	报告 RTDX 软件的版本号

(1) apiversion 属性：报告当前使用的 CCS IDE API(应用程序接口)的版本号。用户不能修改此属性值，当更新 API 或 CCS IDE 时，apiversion 属性值也会随之改变。

注意：API 版本与 CCS IDE 版本不一定相同。

(2) boardnum 属性：指定 CCS IDE 连接对象访问的目标板。利用 ccsboardinfo 函数或 CCS 配置程序来得到目标板的索引号。CCS 配置程序会给每一个安装在主机系统上的目标板分配一个索引号(从 0 开始)。

(3) ccsappexe 属性：包含 CCS IDE 可执行文件 cc_app.exe 的路径。当利用 ccsdsp 函数来创建一个连接对象时，MATLAB 会决定 CCS IDE 可执行文件的路径，并把此路径放在 ccsappexe 属性中。这是一个只读属性，用户不能进行设置。

(4) numchannels属性：报告某一RTDX对象所打开的RTDX通道数目。每打开一个新RTDX通道，numchannels就会增加1。对于一个新创建的连接对象，在为此连接对象打开一个RTDX通道前，numchannels一直为0。

例

```
cc=ccsdsp;
rx=cc.rtdx;
open(rx, 'ichan', 'r', 'ochan', 'w');
get(cc.rtdx)
ans=
     numChannels: 2
            Rtdx: [1x1 COM]
     RtdxChannel: {''  ''  ''}
        ProcType: 103
         Timeout: 10
```

上例中，利用ccsdsp函数来创建一个新的连接对象，并利用open函数为此对象打开两个RTDX通道(一个读通道、一个写通道)，然而再利用get函数显示RTDX对象的属性，可以看到属性numchannels的值变为2。

(5) page属性：包含存储器页。如果用户没有指定page的属性值，它就会包含缺省(默认)的属性值。存储器访问函数需要利用此属性值作为输入参数来访问目标DSP的存储器。

(6) procnum属性：指定CCS IDE连接对象访问的DSP。当创建一个新连接对象时，利用procnum属性来指定连接的目标DSP。CCS设置程序为每一目标板上的DSP都分配一个索引(从0开始)。可以利用ccsboardinfo或CCS设置程序来得到目标DSP的索引号。

要访问某一DSP，两个属性boardnum和procnum都必须指定好。

(7) rtdx属性：是CCS IDE连接对象的一个成员，代表CCS IDE连接对象的RTDX部分，而且rtdx具有自己的属性，可以利用set函数来设置其属性值。

(8) rtdxchannel属性：是一个单元阵，包含RTDX通道名、通道句柄和通道操作模式(读或写)。对每一打开的RTDX通道，rtdxchannel包含如下三个成员：

.rtdxchannel{i,1}：第i通道名
.rtdxchannel{i,2}：第i通道句柄
.rtdxchannel{i,3}：第i通道操作模式，'r'为读，'w'为写

(9) timeout属性：指定CCS IDE等待任何过程完成所需的最大时间。连接对象中有两种timeout：一个是全局的，另一个是局部的。可以在创建连接对象时设置全局timeout属性值，默认的timeout属性值为10 s。而当利用函数向CCS IDE或目标DSP读写数据时，可以设置局域timeout值而忽略掉全局timeout值。如果在读写操作中没有指定timeout值，则此操作函数就会利用全局timeout值。

(10) version属性：报告当前使用的RTDX软件的版本号。用户不能改变此属性值。当更新API或CCS IDE时，version属性值会随之改变。

5.3　CCSLink 嵌入式对象

CCSLink 提供的连接对象可以直接访问目标 DSP 的存储器和寄存器，这对于开发嵌入式数字信号处理应用程序无疑是一种强大的工具。而对于用 C/C++语言开发的应用程序，如果能够按访问 C 符号(C 变量或常量等)的方式来访问存储器中的数据，将会更加方便。

CCSLink 的嵌入式对象提供了这种功能，利用嵌入式对象可以代表和访问嵌入在目标 DSP 中的 C 变量和数据。CCSLink 提供了多个函数(例如：createobj、convert、write)来创建嵌入式对象、访问和操作对应的 C 变量和数据，所有这些操作都在 MATLAB 下完成。

CCSLink 中的连接对象和嵌入式对象都是利用面向对象技术进行操作的。连接对象把 MATLAB 和硬件目标连接在一起，嵌入式对象对目标 DSP 中的所有 C 符号进行访问(读写)和操作(改变数值类型等)。

CCSLink 提供了多种嵌入式对象类型，表 5.2 列出了 CCSLink 提供的所有嵌入式对象类型及其对应的属性和操作函数。接下来再详细介绍嵌入式对象的所有属性。嵌入式对象的操作函数在 5.4 节中再作详细介绍。

表 5.2　嵌入式对象列表

嵌入式对象类型名	属　性	操作函数	描　　述
Bitfield	name,address,bitsperstorageunit, numberofstorageunits,link,timeout	copy,disp, read, write	对位域内容进行描述和访问
Enum	name,address,bitsperstorageunit, numberofstorageunits,link,timeout	equivalent	对存储器中的枚举类型数据进行描述和访问
Numeric	arrayorder,binarypt,size,storageunitsper value, name,address,bitsperstorageunit, numberofstorageunits	cast, convert, reshape	对存储器中的数值数据进行描述和访问
Pointer	name,address,bitsperstorageunit, numberofstorageunits,link,timeout	deref	对存储器中的指针类型数据进行描述和访问
Renum	name,regname,numberofstorageunits, link,timeout	equivalent,write	对寄存器中的枚举类型数据进行描述和访问
Rpointer	name,regname,numberofstorageunits, link,timeout	deref,read,write	对寄存器中的指针类型数据进行描述和访问
Rstring	name,regname,numberofstorageunits, link,timeout	equivalent	对寄存器中的字符串内容进行描述和访问
String	name,address,bitsperstoragenuit, numberofstorageunits,link,timeout	equivalent	对存储器中的字符串内容进行描述和访问
Structure	name,filename,address,type,savedregs, variables,inputvars,outputvars,link, timeout	getmember,read,write	对存储器中的结构体进行描述和访问
Rnumeric	name,binarypt,regname,numberofstorag eunits,link,timeout	cast,convert,reshape, read,write	对寄存器中的数值数据进行描述和访问

(1) address 属性：报告嵌入式对象所访问的 C 符号的首地址——存储器地址或寄存器名。在某些情况下，如果 DSP 支持存储器页，则地址以[偏移量　存储器页]格式给出。

如果用户改变此属性的偏移量值或存储器页值，则此对象就指向存储器的不同位置。改变 address 属性并不会影响 C 符号本身所在的地址。

(2) arrayorder 属性：指定嵌入式对象把线性存储空间数据解释为矩阵的列还是行。

arrayorder 属性值为两种字符串——'row－major' (C 语言习惯)或'column－major' (MATLAB 习惯)。例如：假定存储器中有 9 个数值：1，2，…，9，把这 9 个数值读到一个 3×3 的矩阵中，则不同 arrayorder 值的结果分别为：

```
row－major   1  2  3            column－major  1  4  7
             4  5  6                           2  5  8
             7  8  9                           3  6  9
```

(3) binarypt 属性：指定某位的二进制小数点的位置。要正确解释存储器中的某一数值，需要知道它的数据类型和小数点的位置，才能把它从二进制或十六进制表示转换为十进制表示。对于一个定点数据类型，其值由字长、二进制小数点的位置和是否为符号数来决定。定点数值利用二进制小数点的位置来进行定标和解释。

(4) bitsperstorageunit 属性：报告目标 DSP 的存储器位数(每地址单元)。例如，8、16、32。

(5) endianness 属性：指定如何解释存储器中的位组合格式(端口)：little－endian 还是 big－endian。big－endian 格式指定字的最低位(LSB)在高地址空间上；little－endian 指定字的最低位在低地址空间上。

此属性值为两种字符串 'little'或'big'。

例如创建一个数值对象来访问存储器中的 ddat 变量的过程如下：

```
ddat=createobj(cc, 'ddat')
 get(ddat)
                address: [40072 0]
     bitsperstorageunit:   8
   numberofstorageunits:   32
                   link :  [1×1 ccsdsp]
                timeout:   10
                   name:   'ddat'
               wordsize:   64
 storageunitspervalue: 8
                   size:   4
             endianness:   'little'
             arrayorder:   'row–major'
                 prepad:   0
                postpad:   0
              represent:   'float'
               binarypt:   0
```

从上例可以看到 endianness 的值为 'little'。

(6) isrecursive 属性：指示是否指针指向了自己。

(7) label 属性：包含枚举类型对象元素名的单元矩阵。

(8) link 属性：指定创建嵌入式对象时用到的连接对象。此属性值包含连接对象的名。

例

```
cvar=createobj(cc, 'idat')
cvar.link
  CCSDSP Object:
    API version : 1.2
    Processor type : TMS320C6711
    Processor name : CPU_1
    Running? : No
    Board number : 0
    Processor number : 0
    Default timeout : 10.00 secs
    RTDX channels : 0
```

(9) member 属性：返回一个标识 MATLAB 结构体对象的句柄，此结构体包含嵌入式对象所访问结构体中的所有成员。

例如，假定 DSP 存储器中有一如下定义的结构体：

```
typedef enum TAG_myEnum {MATLAB = 0, Simulink, SignalToolbox, MATLABLink,
                                    EmbeddedTargetC6x} myEnum;
 struct TAG_myStruct {
 int iy[2][3];
 myEnum iz;
}myStruct = { {{1,2,3},{4, - 5,6}}, MATLABLink};
```

然后创建一个结构体对象 cvar 来访问此结构体，显示对象 cvar 的属性如下：

```
get(cvar)
        name: 'myStruct'
        member: [1×1 ccs.containerobj]
        membname: {'iy'    'iz'}
        memoffset: [0    24]
        address: [40032    0]
        storageunitspervalue: 28
        arrayorder: 'row-major'
```

可以看到，member 属性返回一个名为 ccs.containerobj 的结构体对象句柄。

(10) membname 属性：包含被嵌入式对象所访问结构体中的所有成员名。例如假定目标 DSP 中有一结构体：

```
struct   tag {int a; int B; int b; } var;
```

创建一个结构体对象来访问此结构体，然后显示结构体对象的 membname 属性：

```
var = createobj (cc, 'var')
get (var, 'membname')
  'a'  'B'  'b'
```

(11) memoffset 属性：指定结构体中每一成员相对于结构体首地址的偏移量。memoffset 是一向量，向量的长度为结构体成员的数目，向量的第一个元素值总为零，向量的第二个元素值为结构体第二个成员相对于首地址的偏移量，以此类推。

例

```
cvar = createobj(cc, 'myStruct');
get(cvar.memoffset)
          ans=
             [0   24]
```

从上例可以看到，结构体对象的memoffset属性值为[0 24]。

(12) name 属性：提供 C 符号或嵌入式对象名。

(13) numberofstorageunits 属性：给出被访问的 C 符号总共所占据的地址单元数。

(14) page 属性：指定被访问 C 符号的存储器页。对于不支持存储器页的 DSP，例如 C6701，page 属性值总是为 0。当用 get 获取对象的 address 属性时，其地址以[地址，存储器页]格式给出。

(15) postpad 属性：指定缓冲器尾端所需要的填充位数(填满缓冲器)，用来决定缓冲器中最后一个数值(忽略掉填充位)。

(16) prepad 属性：指定缓冲器首端所需要的填充位数(填满缓冲器)。

(17) represent 属性：包含一字符串，用来指定被访问 C 符号的数据类型。represent 属性指定 MATLAB 如何来解释存储器中的数据。

represent 包含的属性值如表 5.3 所示。

表 5.3 represent 的属性值

属性值	描 述	代表的 MATLAB 中数据类型	C5xDSP 字长限制	C6xDSP 字长限制
'float'	IEEE 浮点数表示，32 或 64 位	'double', 'single', 'long double', 'float'	32, 64 位	32, 64 位
'fract'	有符号分数定点数	—	—	—
'signed'	有符号二进制补码整数	'int32','int16','int8','long', 'int','char', 'Q15'	16, 24, 32, 40, 48, 56, 64 位	8, 16, 24, 32, 40, 48, 56, 64 位
'ufract'	无符号分数定点数			
'unsigned'	无符号二进制补码整数	'uint32','unsigned int', 'Q0.31'	16, 24, 32, 40, 48, 56, 64 位	8, 16, 24, 32, 40, 48, 56, 64 位
'binary'	二进制数	—	16, 24, 32, 40, 48, 45, 64 位	8, 16, 24, 32, 40, 48, 56, 64 位

(18) size 属性：返回所访问的数值矩阵的维数。size 为一向量，其长度为所访问符号的矩阵维数，向量的每一元素值报告此维的长度。

例如，程序代码中有如下变量定义：

```
int x[3][2] ={(1, 2), (3, 4), (5, 6)};
```

创建一个数值对象来访问程序中的变量 x，并用 get 函数显示对象的 size 属性值：

```
x = createobj (cc, 'x');
 get (x, 'size')
  ans=
      [3   2]
```

上例中，size 的属性值为[3　2]，表示 3×2 矩阵。

(19) storageunitspervalue 属性：指定所访问 C 符号的每一元素值所占据的存储单元数目。

例

```
cfield = getmember(cvar,'iz');
  get(cfield.storageunitspervalue)
   ans=
      4
```

上例表明'iz'的每一元素值所占据的存储单元数为4。

(20) typestring 属性：返回指针所指向的数据类型。

例如，对于一个访问浮点类型指针的嵌入式对象，返回的 typestring 属性值为'float*'。

(21) value 属性：报告与枚举类型的元素相对应的数值，value 值为一向量，其长度等于枚举类型的长度，向量的元素就是所对应的值。

例

```
get(cfield. value)
  ans=
     [0    1    2    3    4]
  get(cfield. label)
      ['MATLAB'  'Simulink'  'SignalToolbox'  'MATLABLink'  'EmbeddedTargetC6x']
```

从上例可以看到，value属性值和label属性值的对应关系。

(22) wordsize 属性：指定所访问 C 符号的字长(位数，通常为 8、16 或 32)。

5.4　CCSLink 函 数

CCSLink 中提供了多种函数来对 CCS IDE 连接对象、RTDX 连接对象和嵌入对象进行操作。表 5.4 中列出了对 CCS IDE 连接对象进行操作的函数，表 5.5 中列出了对 RTDX 连接对象进行操作的函数，表 5.6 为嵌入式对象操作函数。本节详细介绍每一函数的功能和使用方法。

表 5.4 CCS IDE 连接对象操作函数

函数名	描述
activate	激活 CCS IDE 中的某一个文件、工程或编译链接配置
add	向 CCS IDE 当前的工程中添加一个文件
address	返回符号的地址和存储器页
animate	运行目标 DSP 上的应用程序，到达断点(breakpoint)处后更新 CCS IDE 窗口，然后继续运行
build	编译链接 CCS IDE 中当前的工程
ccsboardinfo	返回 CCS IDE 识别的目标板(或软件模拟器 Simulator)及 DSP 信息
ccsdsp	创建一个 CCS IDE 的连接对象
cd	改变 CCS IDE 的工作路径
clear	清除与 CCS IDE 的连接
close	关闭 CCS IDE 中打开的文件
delete	删除 CCS IDE 文件中的调试点
dir	列出 CCS IDE 当前工作目录中的文件
disp	显示 CCS IDE 连接对象的属性
display	显示 CCS IDE 连接对象的属性
get	获取 CCS IDE 连接对象的属性值
goto	定位程序计数器(PC)到指定的程序代码位置
halt	终止目标 DSP 上某一过程的执行
info	返回目标 DSP 的信息
insert	在源文件或地址处加入一个调试点
isreadable	确定 MATLAB 是否可以读指定的存储器段
isrtdxcapable	确定目标 DSP 或板是否支持 RTDX
isrunning	测试目标 DSP 是否正在执行程序
isvisible	测试 CCS IDE 是否正在桌面上运行
iswritable	确定 MATLAB 是否可以向指定的存储器段写
list	从 CCS 中返回各种信息列表
load	把一个可执行文件(*.out)加载到目标处理中
new	在 CCS IDE 中创建并打开一个新文本文件、工程文件或编译链接配置
open	加载一个文件到 CCS IDE 中
profile	返回程序的统计剖析信息
read	从目标 DSP 的指定存储器中读取数据
regread	从目标 DSP 的指定寄存器中读取一个值
reload	重新向目标 DSP 加载最近加载的程序文件
regwrite	向目标 DSP 的指定寄存器中写入一个值
remove	从 CCS IDE 当前的工程中删除一个文件
reset	复位目标 DSP
restart	把程序计数器(PC)复位到当前程序的入口处
run	运行目标 DSP 中的程序
save	保存 CCS IDE 中的文件或工程
set	设置对象属性
symbol	从 CCS IDE 中返回最近加载程序的符号表
visible	设置 CCS IDE 窗口是否在桌面上可见
write	向目标 DSP 的存储器写入数据

表 5.5　RTDX 连接对象操作函数

函 数 名	描　　述
clear	清除与 RTDX 的连接
close	关闭某一打开的 RTDX 通道
configure	配置 RTDX 通道缓冲器
disable	禁止 RTDX 接口、某一指定的通道或所有 RTDX 通道
disp	显示 RTDX 连接对象的属性
display	显示 RTDX 连接对象的属性
enable	使能 RTDX 接口、某一指定的通道或所有 RTDX 通道
flush	冲刷某一或多个指定 RTDX 通道的数据或信息
get	获取 RTDX 连接对象的属性值
info	返回指定 RTDX 连接对象的属性值
isenabled	确定 RTDX 接口或 RTDX 通道是否已使能
isreadable	确定 MATLAB 是否可以读指定的 RTDX 通道
iswritable	确定 MATLAB 是否可以向指定的 RTDX 通道写
msgcount	返回 RTDX 读通道中信息的数目
open	打开一个 RTDX 通道
readmat	从指定的 RTDX 通道中把数据读入矩阵中
readmsg	从指定的 RTDX 通道中读信息
set	设置 RTDX 对象属性
writemsg	向指定的 RTDX 通道写入信息

表 5.6　嵌入式对象操作函数

函 数 名	描　　述
addregister	添加寄存器到保留寄存器列表中，这些保留的寄存器位于 savedregs 属性中
assignreturnstorage	给函数的输出结果分配一个存储空间
cast	在 MATLAB 中改变对象的数据类型，不影响目标 DSP 中所访问 C 符号的数据类型
cexpr	在目标 DSP 上执行 C 或 GEL 语言表达式
convert	改变对象的 represent 属性，使其从一种数据类型转变到另一种数据类型
copy	复制一个对象
Createobj	创建一个嵌入式对象来代表嵌入在目标 DSP 程序中的 C 变量、数据或函数
deleteregister	从保留寄存器列表中删除某一或多个寄存器
deref	返回一个对象，此对象代表指针所指向的内容
equivalent	返回与输入参数等价的字符串或数值
execute	在目标 DSP 上执行一个函数
getmember	返回一个代表结构体成员的对象
readnumeric	读一个字符串对象并把它换成对应的 ASCII 数值
reshape	改变矩阵的形状

(1) activate 函数：

语法形式：

activate (cc, 'objectname', 'type')

说明：此函数用来激活 CCS IDE 中由 objectname 和 type 指定的文本文件、工程或编译链接配置。cc 为创建的 CCS IDE 连接对象句柄。type 可以是如下三种字符串：

- 'project'：使 CCS IDE 中的某一工程成为当前激活工程，objectname 必须具有 .pjt 扩展名；
- 'text'：使某一指定的文本文件成为 CCS IDE 文档窗中当前的文件，objectname 必须具有扩展名；
- 'buildcfg'：使 CCS IDE 中某一指定编译链接配置成为当前配置。

例如，在 CCS IDE 中创建两个工程：

```
cc=ccsdsp;                          %建立连接对象
visible (cc, 1);                    %使 CCS 在桌面上可见
new (cc, 'myproject1.pit');         %创建新工程
new (cc, 'myproject2.pit');         %创建新工程
```

激活 myproject1 工程，使其成为 CCS IDE 中的当前工程，再向此工程中添加一源文件 C6701evm_adc.c，并使此源文件成为文档窗口中的当前文件。其 MATLAB 代码如下：

```
activate ( cc, 'myproject1.pjt');            %激活工程
add ( cc, 'C6701evm_adc.c');                 %向工程加入文件
activate ( cc, 'C6701evm_adc.c', 'text');    %激活文本文件
```

(2) add 函数：

语法形式：

add (cc, 'filename')

说明：把某一指定的文件 filename 添加到 CCS IDE 的当前工程中。filename 可以是如下文件类型：

- 源文件：.c, .cpp, .cc, .ccx, .sa, .a, .s。
- 目标和库文件：.o, .lib。
- 链接命令文件： .cmd。
- DSP/BIOS配置文件：.cdb
- visual Linker Recipe 文件：.rcp。

(3) addregister 函数：

语法形式：

addregister (ff, regname)

addregister (ff, regnamelist)

说明：把程序中使用到的寄存器 regname 或寄存器列表 regnamelist 添加到保留的寄存器列表中。

ff 为函数对象句柄，当此函数完成运行后，保留的寄存器就会恢复到原来的初始状态。当利用 createobj 来产生一个函数句柄时，编译器会产生默认的保留寄存器。对于不同的 DSP，这些默认的保留寄存器不同，如下所示：

C54x DSP：AR1，AR6，AR7，SP(由 MATLAB 要求，而不是由编译器规定)。

C62x DSP：A0，A2，A6，A7，A8，A9，B0～2，B4～9，B15 (MATLAB 要求)。A3，A4，A5，B3 在函数调用过程中，也必须保留它们。

C64x DSP：A0，A2，A6，A7，A8，A9，A16～31，B0～2，B4～9，B16～31，B15(MATLAB 要求)。A3，A4，A5 在函数调用过程中，也必须保留它们。

(4) address 函数：

语法形式：

```
a=address (cc, 'symbolstring')
```

说明：返回符号 symbolstring 的存储器地址和存储器页。a 为单元阵，第一行为符号名的字符串，第二行为[地址，页]。

例如，从目标 DSP 中的'ddat'符号处读入 4 个数值，并作为双精度数分配给 ddatv；

```
ddatv=read (cc,address (cc, 'ddat'), 'double', 4 )
```

(5) animate 函数：

语法形式：

```
animate (cc)
```

说明：开始执行目标 DSP 中的程序，直到遇到一个断点(breakpoint)后，停止程序的执行并更新 CCS IDE 中的所有窗口，然后继续执行程序，如此循环下去直到用户停止它的执行。

(6) assignreturnstorage 函数：

语法形式：

```
assignreturnstorage(ff, address)
assignreturnstorage(ff, handle)
```

说明：设置函数对象的 outputvar 属性值，即为函数返回的结果(结构体)分配一个存储空间。address 为地址值，handle 为结构体对象的句柄。

(7) build 函数：

语法形式：

```
build(cc,timeout)
build(cc)
build(cc, 'all ', timeout)
build(cc, 'all')
```

说明：编译链接 CCS IDE 中的当前工程。timeout 为此编译链接过程的超时设定，单位为秒，当编译链接过程超过 timeout 的值后，MATLAB 就会返回出错信息。如果省略 timeout，则 build 过程会利用 CCS IDE 中的全局 timeout 值作为默认超时设置。'all'为重新编译链接当前工程中的所有文件，否则只编译链接从上一次编译链接完成后改变过的文件。

(8) cast 函数：

语法形式：

```
objname2 =cast(objname, datatype)
objname2 =cast(objname, datatype, size)
```

说明：复制 objname 对象到 objname2 对象中，并且 objname2 对象的 represent 属性值

改变为 datatype 指定的数据类型，objname2 对象的 size 属性值改变为由 size 指定的值，objname 对象的属性值不变。datatype 可选的数据类型如表 5.3 所示。cast 的详细使用例子，在 5.5 节中再作介绍。

(9) ccsboardinfo 函数：

语法形式：

```
ccsboardinfo
boards =ccsboardinfo
```

说明：返回 CCS IDE 识别的目标板及其 DSP 信息。此函数在创建连接对象时经常用到，我们在前面已作了介绍，这里不再详述。

第二种语法形式是把返回的目标板配置信息放入到结构体矩阵 boards 中，boards 的每一元素为对应目标板的名称、索引号和 proc 结构体，proc 结构体中包含此目标板上每一个 DSP 的信息。

例如，假定主机上装有两个目标板，其中一个为 simulator，另一个为 C6211DSK，而且此目标板上有三片目标 DSP。利用 ccsboardinfo 可以显示如下信息：

```
Board   Board                                Proc   Processor      Processor
Num     Name                                 Num    Name           Type
---------------------------------------------------------------------------------
1       C6xxx Simulator (Texas Instrum ..)   0      CPU            TMS320C6211
0       C6211 DSK (Texas Instruments)        2      CPU_3          TMS320C6x1x
0       C6211 DSK (Texas Instruments)        1      CPU_4_1        TMS320C6x1x
0       C6211 DSK (Texas Instruments)        0      CPU_4_2        TMS320C6x1x
```

创建一个连接对象，访问 0 号目标板上的 2 号 DSP(CPU_3)：

```
boards=ccsboardinfo;
cc=ccsdsp('boardnum', boards(2).number, 'procnum', boards(2).proc(1).number);
```

要注意 boards 和上述信息列表的对应关系。

(10) ccsdsp 函数：

语法形式：

```
cc =ccsdsp
cc =ccsdsp('propertyname', 'propertyvalue', ...)
```

说明：创建一个 CCS IDE 连接对象，并把此连接对象的句柄放入 cc 中。MATLAB 利用此句柄与目标 DSP 进行通信。可以创建默认属性的连接对象(第一种语法形式)，也可以在创建连接对象时设置对象属性(第二种语法形式)。连接对象的属性已经在 5.2 节中作了详细介绍。

(11) cd 函数：

语法形式：

```
cd(cc,'directory')
wd =cd(cc, 'directory')
cd(cc, pwd )
```

说明：改变 CCS IDE 的当前工作路径为由 directory 指定的路径，并可以把当前的工作路径返回到 wd 中(第二种语法形式)。第三种语法形式为调用 MATLAB 的函数 pwd，并改

变当前 CCS IDE 工作路径为 pwd 返回的路径。

(12) cexpr 函数：

语法形式：

```
result =cexpr(cc, 'expression', timeout)
result =cexpr(cc, 'expression')
```

说明：在目标 DSP 上执行 expression 指定的表达式，并返回处理结果。expr 的功能类似于 CCS IDE 的命令行对话框。timeout 为超时设置(单位为秒)，当此操作超过 timeout 时，MATLAB 会返回错误信息。expression 可以是 C 表达式、GEL 函数或 C 和 GEL 的混合表达式。

如果 expression 是 C 表达式，则 cexpr 返回一个结果到 MATLAB 中。

如果 expression 是 GEL 函数，则 cexpr 不返回结果。

例　result=cexpr(cc, 'x.a')

返回目标 DSP 中 x.a 的值到 MATLAB 的 result 中。

```
result=cexpr(cc, 'x.b=10')
```

设置目标 DSP 中的 x.b 为 10，并返回 result=10。

```
result=cexpr(cc, 'StartUp( )')
```

在目标 DSP 上执行 GEL 函数 StartUp( ), result 为 NULL，因为 GEL 函数不产生返回值。

```
result=cexpr(cc, ['x.c[2]='int2str(z)])
```

int2str(z)为 MATLAB 函数，即用此函数的结果给目标 DSP 中的 x.c[2]赋值。

(13) clear 函数：

语法形式：

```
clear(cc)
clear('all')
```

说明：clear(cc)为清除与 cc 相关的连接对象。clear('all')清除 MATLAB 中的所有存在的 CCS IDE 和 RTDX 连接对象。利用连接对象进行调试的最后一步都要清除掉不需要的连接对象，以免会带来无法预料的结果。

(14) close 函数：

语法形式：

```
close(cc, 'filename', 'type')
close(rx, 'channel1', 'channel2', …)
close(rx, 'channel')
```

说明：关闭掉 CCS IDE 中由 filename 和 type 指定的文件，type 为文件类型，可以为：

- 'project'：工程文件；
- 'text'：文本文件。

rx 为 RTDX 连接对象句柄，关闭掉 channel1、channel2…等指定的 RTDX 通道。如果 channel 为 all，则关闭掉 rx 中所有打开的通道。

(15) configure 函数：

语法形式：

```
configure(rx, length, num)
```

说明：配置 RTDX 通道缓冲器的大小和数目，length 为每一通道缓冲器的大小(字节表

示)，num 为通道缓冲器的数目。

缓冲器的大小至少为 1024 个字节。由最大的信息长度决定。对于 16 位 DSP，缓冲器的长度至少超过最大信息 4 个字节；对于 32 位 DSP，缓冲器的长度至少超过最大信息 8 个字节。CCS IDE 给每一个 DSP 分配一个缓冲器，而与 num 值无关。

(16) convert 函数：

语法形式：

```
convert(objname, datatype)
convert(objname, datatype, size)
```

说明：改变对象 objname 的 represent 属性值为 datatype 指定的数据类型, size 属性值为 size 指定的值。

datatype 可以指定的数据类型如表 5.3 所示。

(17) copy 函数：

语法形式：

```
objname2 =copy(objname)
```

说明：复制对象 objname 到对象 objname2 中。

(18) createobj 函数：

语法形式：

```
objname =createobj(cc, 'symbolname')
objname =createobj(cc, 'symbolname', 'option')
```

说明：创建一个嵌入式对象 objname，此嵌入式对象代表目标 DSP 中的 symbolname 符号，而且 symbolname 也可以是目标程序代码中的一个函数名。

option 允许用户声明 symbolname 代表一个静态(static)变量还是一个全局(global)变量。

(19) delete 函数：

语法形式：

```
delete(cc, addr, 'type')
delete(cc, addr)
delete(cc, filename, line, 'type')
delete(cc, filename, line)
```

说明：从存储器地址 addr 或源文件 filename 的指定行 line 删除一调试点。addr 只能是地址值，不能是 C 函数、C 表达式或符号。type 为调试点的类型，可以有如下形式：

- 'break'：断点(breakpoint)。如果 type 省略，则默认为此类型。
- ' '：同'break'。
- 'probe'：探点(probepoint)。

如果指定的位置处没有调试点或调试点类型不对，则 MATLAB 会返回一个错误信息。

(20) deleteregister 函数：

语法形式：

```
deleteregister(ff, 'regname')
deleteregister(ff, 'reglist')
```

说明：从函数调用过程中保留的寄存器列表中删除一个寄存器(regname)或多个寄存器

的列表(reglist)。可以删除默认保留寄存器外的所有保留的寄存器。默认保留寄存器详见 addregister 函数。

(21) deref 函数：

语法形式：

```
objname2 =deref(objname)
```

说明：创建一个对象 objname2，此对象代表指针对象 objname 所指向的内容。deref 可以理解为 C 语言中的*。

(22) dir 函数：

语法形式：

```
dir(cc)
```

说明：列出当前 CCS IDE 工作路径中所有的文件和目录。

(23) disable 函数：

语法形式：

```
disable(rx,' channel')
disable(rx, 'all ')
disable(rx)
```

说明：禁止 RTDX 连接对象 rx 中某一打开的 RTDX 通道('channel')，或禁止所有打开的 RTDX 通道('all')，或禁止 RTDX 接口(第三种语法形式)。

使用 disable 必须满足如下要求：

- 当利用 disable 来禁止 RTDX 通道或接口时，目标 DSP 必须正在运行程序。
- 必须已经使能(enable)RTDX 接口。

(24) disp 函数：

语法形式：

```
disp(rx)
disp(cc)
```

说明：显示 RTDX(默认)或 CCS IDE 连接对象的属性名及其属性值。

(25) display 函数：

语法形式：

```
display(cc)
display(rx)
```

说明：同 disp，显示 RTDX 或 CCS IDE 连接对象的属性名及其属性值。

(26) enable 函数：

语法形式：

```
enable(rx, 'channel')
enable(rx, 'all')
enable(rx)
```

说明：使能 RTDX 连接对象 rx 中某一打开的 RTDX 通道('channel')，或使能所有打开的 RTDX 通道('all')，或使能 RTDX 接口(第三种语法形式)。

使用 enable 必须满足如下条件：

- 当使能 RTDX 接口时，目标 DSP 必须正在运行程序。
- 在使能 RTDX 通道前必须先使能 RTDX 接口。
- 通道在使能之前必须先打开。
- 当利用目标程序来使能 RTDX 通道时，会优先于通道的禁止状态。

例如要创建一个 RTDX 连接，使能 RTDX 接口，打开并使能一个输入通道，可使用如下语句：

```
cc =ccsdsp;
enable(cc.rtdx);
open(cc.rtdx, 'inputchannel', 'w');
enable(cc.rtdx, 'inputchannel');
```

(27) equivalent 函数：

语法形式：

```
value =equivalent(objname, input)
```

说明：把输入参数(input)转换成等价的字符串或数值值。如果 input 为字符串，则 value 为对应的 ASCII 值；如果 input 为数值，则 value 为转换成的字符串。input 可以为单个字符串，也可以是数值或字符串矩阵，甚至为数值或字符串单元阵。objname 可以为数值对象、字符串对象、寄存器字符串对象或枚举对象。转换过程根据对象的 charconversion 属性值来进行。

(28) execute 函数：

语法形式：

```
execute(ff)
execute(ff,input1,value1,…,inputn,valuen)
```

说明：在目标 DSP 上执行由句柄 ff 指定的函数。如果没有指定函数的输入参数，execute 利用函数对象中的 inputvars 属性值作为输入参数。函数执行完成后把返回结果放入 outputvar 属性中。

利用 execute 运行函数之前，必须先利用 goto 函数来定位程序计数器(PC)到函数的开始位置。execute 从程序计数器的当前位置开始执行，而不去搜索此函数。并且在运行此函数之前，必须先设置好函数对象的 outputvar 属性。

如果指定了函数的输入参数(第二种语法形式)，则先把输入参数值写入到 inputvars 属性中，再运行此函数。

(29) flush 函数：

语法形式：

```
flush(rx, 'channel', num, timeout)
flush(rx, 'channel', num)
flush(rx, 'channel', [ ], timeout)
flush(rx, 'channel')
```

说明：从 RTDX 连接对象 rx 的'channel'通道中移除 num 个最旧的信息。timeout 为此操作的超时设定(单位为秒)，如果省略，则此操作利用全局 timeout 设置。如果 num 为[ ]，则此操作移走 channel 通道中的所有信息。

前三种语法形式只对 RTDX 读通道进行操作，最后一种语法形式可以应用于读通道或写通道。应用于写通道时，把写通道中的所有信息发送给目标 DSP。

(30) get 函数：

语法形式：

```
get(cc, 'propertyname')
get(cc)
v =get(cc, 'propertyname')
get(rx, 'propertyname')
get(rx)
v =get(rx)
get(objname, 'propertyname')
get(objname)
v =get(objname)
```

说明：获取连接对象、嵌入对象或其它 MATLAB 对象的属性值，并可以把属性名及其值返回到一个结构体 v 中。propertyname 为属性名。

(31) getmember 函数：

语法形式：

```
objname2 =getmember(objname, membername)
objname2 =getmember(objname, index, membername)
```

说明：创建一个对象 objname2 用来访问结构体对象 objname 的某一成员 membername。对象 objname2 的类型由 membername 的数据类型决定，即数值结构体成员返回数值对象，以此类推。

例如，假定目标程序代码中有如下一个结构体：

```
struct testdeepstr {
    int x_int;
  struct mystructa x_str;
  struct mystructa z_str [2 ];
 }str_recur;
```

创建一个结构体对象来访问 str_recur：

```
structtest=createobj(cc, 'str_recur');
```

创建一个数值对象来访问结构体对象的'x_int'成员：

```
xint=getmember(structtest, 'x_int');
```

在 MATLAB 中可以直接访问此结构体成员：

```
write(xint,2);
read(xint)
ans=
    2
```

(32) goto 函数：

语法形式：

```
goto(cc)
goto(cc, 'functionname')
goto(ff)
goto(ff, 'input1', value1, …, 'inputn', valuen)
```

说明：定位程序计数器(PC)到指定的程序代码位置。functionname 为函数名，指定 CCS 在此函数的开始处设置一个断点，然后开始运行目标 DSP 中的程序直到到达此函数的开始处停止。如果省略 functionname，则 CCS 在主函数的入口处设置一断点，然后开始运行程序，到此断点处停止。ff 为函数对象句柄，定位 PC 到此函数的开始处。如果在 PC 的当前位置和函数 functionname(或函数对象句柄 ff)之间有另一断点，则 goto 会在此断点处返回而没有到达指定函数的入口处，因此在运行 goto 命令之前必须清除所有的断点。第四种语法形式定位 PC 到函数的开始处，并且设置函数的入口参数。

(33) halt 函数：

语法形式：

```
halt(cc, timeout)
```

说明：立即停止目标 DSP 中程序的执行。timeout 为设置此操作的超时，如果省略，则会利用默认的全局 timeout 值。

(34) info 函数：

语法形式：

```
info =info(cc)
info =info(rx)
```

说明：info(cc)返回目标 DSP 的信息，并把返回的结构体分配给一个变量。结构体包含如下成员：

- info.procname：在 CCS IDE 设置程序中定义的 DSP 名。
- info.isbigendian：描述 DSP 的字节组合形式，1=big－emdian，0=little－endian。
- info.family：用三个数字识别 DSP 系列，如 320 为 TI 数字 DSP。
- info.subfamily：用十进制数代表的十六进制的 DSP 身份标识(ID)，以识别 DSP 所属的子系列。例如，dec2hex (info.subfamily)产生'67'，表明此 DSP 属于 C67xx 系列。
- info.timeout：超时设置，默认为 10 s。
- info = info(rx)返回一个单元阵 info，此单元阵中包含 RTDX 连接对象 rx 中打开的所有通道名。

例

```
info(cc)
 ans =
      procname: 'CPU'
      isbigendian: 1
      family: 320
      subfamily: 103
      timeout: 10
 info = info(rx)
 info =
         'chan1'
         'chan2'
```

(35) insert 函数：

语法形式：

```
insert(cc, addr, 'type')
insert(cc, addr)
insert(cc, filename, line, 'type')
insert(cc, filename, line)
```

说明：在存储器的地址 addr 处或源程序 filename 的 line 行处插入一个调试点。addr 为十六进制地址，不能是 C 函数、C 表达式或 C 符号。type 为调试点的类型，可选如下类型：

- 'break'：断点(breakpoint)。
- ' '：断点(缺省)。
- 'probe'：探点(probepoint)。
- 'profile'：统计剖析点。

这些类型我们在 CCS 一章中已作了详细介绍，这里不再详述。

例　insert(cc, 15424, 'break');

在目标DSP的0x15424地址处插入一个断点。

insert(cc, 'volume.c', 47, 'probe');

在 volume.c 文件的第 47 行插入一个探点。

(36) isenabled 函数：

语法形式：

```
isenabled(rx, 'channel')
isenabled(rx)
```

说明：确定是否指定的 RTDX 通道'channel'已经使能。如果已经使能则返回 1，否则返回 0。如果'channel'省略，测试 RTDX 接口是否使能，1=是，0=否。

使用 isenabled，必须满足如下条件：

- 目标 DSP 必须正在运行。
- 在检查单个通道或 RTDX 接口状态之前，必须先使能 RTDX 接口。

(37) isreadable 函数：

语法形式：

```
isreadable(cc, address, 'datatype',  count)
isreadable(cc, address, 'datatype')
isreadable(rx, 'channel')
```

说明：确定 MATLAB 是否可以读指定的(由 cc,address, count 和'datatype'指定)DSP 存储器块。如果可以读，返回 1，否则返回 0。address 指定数据块的开始地址，可以为十进制数也可以为十六进制数。整个存储器地址包括两部份：[偏移量，存储器页]。对于不支持存储器页的 DSP，地址的存储器页为 0，设置连接对象的 page 属性值为 0，就可以只利用偏移量来访问整个存储器地址空间了。对于支持存储器页的 DSP，设置连接对象的 page 属性值为数据块所在的存储器页，address(十进制，十六进制)为偏移量，就可访问此数据块。count 指定读取的数据个数，可以为标量值或一个矢量定义的多维数据块。如果省略 count，默认为 1。假定 count 为一矢量[10　10　10]，则读 1000 个(10*10*10)数据。datatype 为数据类

型，即利用 datatype 决定每一数据占据的存储单元(字节)数目。isreadable 支持如下数据类型：'double', 'int8', 'int16', 'int32', 'single', 'uint8', 'uint16', 'uint32'。

第三种语法形式确定 RTDX 通道'channel'是否配置为读操作，是返回 1，否则返回 0。

(38) isrtdxcapable 函数：

语法形式：

b=isrtdxcapable(cc)

说明：确定 DSP 是否支持 RTDX，是返回 1，否则返回 0。

(39) isrunning 函数：

语法形式：

isrunning(cc)

说明：确定 DSP 是否正在运行程序，是返回 1，否则(停止)返回 0。

(40) isvisible 函数：

语法形式：

isvisible(cc)

说明：确定是否 CCS IDE 正在桌面上运行且窗口已经打开，是返回 1，否则(CCS IDE 没有运行或正在后台运行)返回 0。

(41) iswritable 函数：

语法形式：

iswritable(cc, address, 'datatype', count)

iswritable(cc, address, 'datatype')

iswritable(rx, 'channel')

说明：确定 MATLAB 是否可以向指定的(由 cc, address, 'datatype', count 指定)DSP 存储器块写入数据，是返回 1，否则返回 0。对 address、datatype、count 的解释参看 isreadable 函数的介绍。

第三种语法形式确定 RTDX 通道'channel'是否配置为写操作，是返回 1，否则返回 0。

(42) list 函数：

语法形式：

list(ff, varname)

infolist =list(cc, type, option)

说明：从 CCS 中返回各种信息列表。

第一种语法形式列出函数(由函数对象句柄 ff 指定)局部变量 varname 的信息。

第二种语法形式从 CCS 中读入 type 和 option 指定的信息，并把信息结构体返回到 MATLAB 的 infolist 中。type 指定返回的信息类型，可以有如下选项：

- 'project'：返回 CCS 中的当前工程信息。
- 'variable'：返回目标程序中的变量信息。option 指定变量名或变量名列表，如果 option 省略则返回所有局部变量及全局变量信息。
- 'globalvar'：返回目标程序中的全局变量信息。option 指定全局变量名或全局变量名列表，如果 option 省略则返回所有全局变量信息。
- 'function'：返回工程中的函数信息。option 指定函数名或函数名列表，如果 option 省

略则返回工程中的所有函数信息。

- 'type'：返回目标程序中定义的数据类型信息。option 指定数据类型(包括 struct,enum 和 union)或数据类型列表，如果 option 省略则返回所有定义的数据类型信息。

(43) load 函数：

语法形式：

```
load(cc, 'filename', timeout)
load(cc, 'filename')
```

说明：把指定的可执行文件 filename 加载到目标 DSP 中。filename 只能是经过 CCS 编译链接过的*.out 文件。timeout 指定此操作的超时设置(秒)，如果省略，则默认利用全局 timeout 设置。

(44) msgcount 函数：

语法形式：

```
msgcount(rx,'channel')
```

说明：返回 RTDX 读通道'channel'中未读的信息数目。'channel'只能是 RTDX 读通道不能是写通道。

(45) new 函数：

语法形式：

```
new(cc, 'objectname', 'type')
new(cc, 'objectname')
```

说明：在 CCS IDE 中创建并打开一个新的文本文件、工程文件或编译链接配置。objectname 为创建并打开的新文件名(包括文件路径和扩展名)。type 为创建的类型，可以有如下选项：

- 'text'：创建一个新文本文件(包括.c, .cpp, .cc, .ccx, .sa, .a*, .s*, .o*, .lib, .cmd 文件)。
- 'project'：创建一个新工程(.pjt)。
- 'projext'：创建一个新外部工程，即此工程利用外部 makefile 文件(.pjt)。
- 'projlib'：创建一个新工程库(.lib)。
- [ ]：创建一个新工程。
- 'buildcfg'：在当前工程中创建一个新的编译链接配置。

若 type 省略，则默认为工程文件(.pjt)。

(46) open 函数：

语法形式：

```
open(rx, 'channel1', 'mode1', 'channel2', 'mode2',…)
open(rx, 'channel', 'mode')
open(cc, filename, filetype, timeout)
open(cc, filename, filetype)
open(cc, filename)
```

说明：前两种语法形式为打开新的 RTDX 通道('channel1', 'channel2'…为通道名)，对于每一个通道，根据其后的'mode1', 'mode2'…来配置为读('r')或写('w')通道。

后三种语法形式把文件 filename 加载在 CCS IDE 中。filetype 指定文件类型，可以有如

下选项：

- 'program'：目标 DSP 的可执行程序(.out)。
- 'project'：CCS IDE 工程文件。
- 'text'：文本文件。
- 'workspace'：CCS IDE 工作空间文件(.wks)。

timeout 指定此操作的超时设置(单位秒)，如果省略，则会利用全局 timeout 设置。

(47) profile 函数：

语法形式：

```
ps=profile(cc, 'option', timeout)
ps=profile(cc, 'option')
ps=profile(cc)
```

说明：利用 DSP/BIOS 中的 STS 目标返回程序代码的统计剖析信息。利用 profile 可以帮助开发人员统计程序代码的执行情况。关于 DSP/BIOS 的详细介绍，请参考第 3 章。

option 指定如何返回统计剖析信息，可以有如下选项：

- 'raw'：返回一个无格式的 STS 时间统计信息列表，返回所有与时间相关的对象。
- 'report'：利用 HTML 格式返回与'raw'选项相同的信息，仅适用于包含 DSP/BIOS 的工程。
- 'tic'：返回格式化列表的 STS 统计时间信息。滤除'raw'选项返回的部分信息。返回的对象必须都是与时间相关的，用户定义的对象不返回(可以利用'raw'选项查看用户定义的对象)。

返回的结构体 ps，包含如下成员：

ps.cpuload：CPU 执行时间占总时间的百分比。

ps.sts：工程中定义的 STS 对象矢量。

ps.sts(n).name：用户定义的第 n 个 STS 对象的名字。

ps.sts(n).units：指定第 n 个 STS 对象的定时器：'Hi Time'(高时间分辨率)或'Low Time'(低时间分辨率)。

ps.sts(n).max：第 n 个 STS 对象的最大测量时间(单位秒)。

ps.sts(n).avg：第 n 个 STS 对象平均统计时间(单位秒)。

ps.sts(n).count：第 n 个 STS 对象的测量次数。

(48) read 函数：

语法形式：

```
mem =read(cc,address,'datatype',count,timeout)
mem =read(cc,address,'datatype',count)
mem =read(cc,address,'datatype')
data =read(objname)
data =read(objname,index)
data =read(objname,member,memberindex,structindex)
data =read(…,timeout)
```

说明：前三种语法形式对连接对象进行操作，从目标 DSP 的存储器中读入一段数据(由

address, count 和'datatype'指定)。address,count 和'datatype'的定义在 isreadable 的介绍中已作了详细说明。如果 count 省略，则默认为 1。若 timeout 省略，则默认为利用全局 timeout 设置。

后四种语法形式对嵌入式对象进行操作。

第四种和第五种语法形式从嵌入式对象 objname 所指定的存储器中读入数据并根据 objname 的属性(例如,wordsize,storageunitspervalue,size,represent 和 binarypt)将其转换成一种数值表示。

index 为矩阵的行列号，可以为标量或矢量，它指定读入哪个(或哪些)数据元素：

- 如果 index 为[ ]或省略，则读入 objname 所代表符号的所有数据。
- 如果 index 为标量，则读入所访问符号的第 index 个列矢量。
- 如果 index 为一矢量，则返回 index 指定矩阵中的元素。例如，index 为[2　2　2]，则返回矩阵中的第[2][2][2]元素值。注意，此时 index 矢量的维数必须与对象的数据维数(在 size 属性中指定)一致，即如果对象为四维矩阵，则 index 也必须为四维矩阵。
- 如果要求返回多个元素，则可以利用 MATLAB 中的标准方法，例如，index 为[1:6]则返回前 6 个元素(假定对象所访问的符号为一向量)。

最后两种语法形式是读入结构体成员的某一元素。objname 为结构体对象。member 指定读取的结构体成员名。memberindex 指定读取数据元素的索引号(在 member 成员中)。当读取的结构体为 objname 的一个向量元素或结构体成员时，structindex 用来指定此结构体所在 objname 中的索引号。read 返回的数据类型与读取的结构体成员类型一致。如果 memberindex、member 和 structindex 都省略，则 read 返回 objname 的整个结构体。

timeout 指定此操作的超时设置，如果省略，则利用全局 timeout 设置(cc 的 timeout 属性值)。

(49) readmat 函数：

语法形式：

```
data =readmat(rx,channelname,datatype,siz,timeout)
data =readmat(rx,channelname,datatype,siz)
```

说明：从 RTDX 读通道中读入一个数据矩阵。channelname 指定 RTDX 通道名，通道名必须在目标 DSP 程序中已经定义过，详见第 3 章。

datatype 指定读取的数据类型，支持的数据类型包括：'double','int16','int32','single'和'uint8'。

siz 定义数据矩阵的维数，为一向量。例如，siz 为[5 5]表示读入数据矩阵的维数为 5×5。siz 指定矩阵的元素数必须为整数个信息，即 siz 指定矩阵的元素数必须为 L 的整数倍(L 为每一信息的元素数)。

在读一个 RTDX 通道之前，必须先打开并且使能此通道。

timeout 指定此操作的超时设置，如果省略，则利用全局 timeout 设置。

(50) readmsg 函数：

语法形式：

```
data =readmsg(rx,channelname,datatype,siz,nummsgs,timeout)
data =readmsg(rx,channelname,datatype,siz,nummsgs)
```

data =readmsg(rx,channelname,datatype,siz)

data =readmsg(rx,channelname,datatype,nummsgs)

data =readmsg(rx,channelname,datatype)

说明：从指定的 RTDX 读通道 channelname 中读入 nummsgs 个信息。channelname 为 RTDX 读通道名，通道名必须在 DSP 程序中已经定义过，详见第 3 章。

nummsgs 指定读入的信息数目，如果 nummsgs 为'all'，则从通道中读出当前所有信息；如果 nummsgs 省略，则从通道中读出 1 个信息。

datatype 指定信息的数据类型，支持的数据类型包括：'double', 'int16', 'int32', 'single' 和 'uint8'，所有的信息都具有同样的数据类型。

siz 指定每一数据矩阵的维数，信息中的所有数据元素都读入此矩阵。例如，siz 为[m,n]，numsgs 为 10，则读入 10 个 m×n 数据矩阵，m*n 必须等于每一信息的元素数目 L。如果 siz 省略，则 siz 默认为[1,L]，L 为每一信息的数据元素数目。每一信息可以包括多个数据元素，例如 L 个。

timeout 指定此操作的超时设置，如果省略，则利用全局 timeout 设置。

readmat 和 readmsg 的详细使用方法，我们将在 5.5 节的例子中再介绍。

(51) readnumeric 函数：

语法形式：

data =readnumeric(objname)

data =readnumeric(objname, index)

data =readnumeric(…, timeout)

说明：读入一个字符串(由 objname 字符串对象指定)并把字符转化成对应的 ASCII 值。objname 为字符串对象。index 指定字符串中某一字符的索引号。如果 index 省略，则把 objname 中的所有字符转换成对应的 ASCII 值。

timeout 指定此操作的超时设置，如果省略，则利用全局 timeout 设置。

(52) regread 函数：

语法形式：

reg =regread(cc,'regname','represent',timeout)

reg =regread(cc,'regname','represent')

reg =regread(cc,'regname')

说明：从目标 DSP 的指定寄存器中读入一个数值，并作为双精度浮点值返回到 MATLAB 中。regname 为单个寄存器名或寄存器对(64 位)。

represent 定义 regname 寄存器中数据的格式，有如下选项：

- '2scomp'：二进制补码表示的有符号整数值，当 represent 省略时默认为此格式。
- 'binary'：无符号二进制整数。
- 'ieee'：32 位或 64 位 IEEE 浮点格式。

无论 represent 如何指定寄存器中的数据类型，regread 总把此值转换为双精度浮点值返回到 MATLAB 空间中。

timeout 指定此操作的超时设置，如果省略，则利用全局 timeout 设置。

例如，从C5000 DSP的PC寄存器中读入一个无符号二进制整数，并把此值转换成双精

度浮点数分配给MATLAB中的xreg变量，可以用以下语句实现：

```
xreg=regread(cc, 'PC', 'binary');
```

从C6000 DSP的A0寄存器中读入一个二进制补码表示的有符号整数值，并把此值转换成双精度浮点数分配给MATLAB中的yreg变量，可以用以下语句实现：

```
yreg = cc.regread('A0');
```

(53) regwrite 函数：

语法形式：

```
regwrite(cc,'regname',value,'represent',timeout)
regwrite(cc,'regname',value,'represent')
regwrite(cc,'regname',value,)
```

说明：把一数值(value)写入到 DSP 的指定寄存器中(regname)，写入前把这一数值转换成 represent 指定的数据类型。regname 指定单个寄存器名或寄存器对(64 位)。

represent 指定存入寄存器中的数据格式，有如下选项：

- '2scomp'：二进制补码表示的有符号整数值，当 represent 省略时默认为此格式。
- 'binary'：无符号二进制整数。
- 'ieee'：32 位或 64 位 IEEE 浮点格式。

timeout 指定此操作的超时设置，如果省略，则利用全局 timeout 设置。

例如，可用以下语句写入一个数值 0x100 到 C5000DSP 的 PC 寄存器中：

```
regwrite(cc, 'pc', hex2dec('100'), 'binary');
```

可用以下语句写入一个 64 位的浮点数到 B1:B0 寄存器对中：

```
regwrite(cc, 'b1:b0', 64, 'ieee');
```

(54) reload 函数：

语法形式：

```
s =reload(cc, timeout)
s =reload(cc)
```

说明：把最近加载的可执行文件重新加载到目标 DSP 中。如果加载成功，s 返回程序文件的整个路径，否则 s 返回空。

timeout 指定此操作的超时设置，如果省略，则利用全局 timeout 设置。

(55) remove 函数：

语法形式：

```
remove(cc,'filename')
```

说明：从 CCS IDE 的当前工程删除一个文件(filename)。filename 指定删除的文件名。

(56) reset 函数：

语法形式：

```
reset(cc, timeout)
reset(cc)
```

说明：停止目标 DSP 中程序的执行，异步执行一个 DSP 复位操作，所有寄存器返回到上电时的初始设置。

timeout 指定此操作的超时设置，如果省略，则利用全局 timeout 设置。

(57) reshape 函数：

语法形式：

```
reshape(x,m,n)
reshape(x,m,n,p …)
reshape(x,[m n p …])
reshape(x, …, [ ], …)
```

说明：改变矩阵的形状，但元素数目不变。它与 MATLAB 中的 reshape 函数相同。x 为一矩阵，按此矩阵的列把它变成一个 m×n×p…维的矩阵，x 矩阵的元素数目必须与 m*n*p…相等。

第二种语法形式和第三种语法形式的含义相同，都是把 x 矩阵变成 m×n×p…维矩阵。

最后一种语法形式中的[ ]要求 MATLAB 根据元素数目相等原则来计算此维的长度，并用此长度来代替[ ]。在 reshape 函数的输入参数中只能有一个[ ]。

(58) restart 函数：

语法形式：

```
restart(cc,timeout)
restart(cc)
```

说明：立即停止目标 DSP 并复位程序计数器(PC)到当前程序的入口点。restart 只是复位程序计数器(PC)而不运行程序，需要利用 run 函数来运行程序。

timeout 指定此操作的超时设置，如果省略，则利用全局 timeout 设置。

(59) run 函数：

语法形式：

```
run(cc, 'state', timeout)
```

说明：从程序计数器 PC 的当前位置开始执行目标 DSP 中的程序。state 决定用户何时得到程序控制，有如下三种选项：

- 'run'：开始执行程序，程序运行后就把控制权返回到 MATLAB 中。因此在 DSP 运行过程中，可以继续在 MATLAB 下进行工作。当 state 省略时，此项为默认设置。
- 'runtohalt'：开始执行程序，程序运行后直到程序碰到一个断点或程序执行结束后才会返回控制权。
- 'tohalt'：改变运行过程的状态为'runtohalt'，当 DSP 停止后才返回控制权。如果 DSP 已经停止，则此 state 设置会令 run 立即返回。

timeout 指定此操作的超时设置，如果省略，则利用全局 timeout 设置。

(60) save 函数：

语法形式：

```
save(cc, 'filename','type')
save(cc, 'filename')
```

说明：保存 CCS IDE 中的文件或工程。filename 指定要保存的文件名。如果 filename 为'all'，则保存 type 指定类型的所有文件；如果 filename 为[ ]，则保存 type 指定类型的当前文件。type 指定保存的文件类型，可以有如下两种选项：

- 'project'：工程文件(.pjt)。
- 'text'：文本文件(.a*, .c, .cc, .ccx, .cdb, .cmd, .cpp, .lib, .o*, .rcp 和 .s*)。

例　save(cc,'all','project')　　保存 CCS IDE 中所有打开的工程
　　save(cc,'my.pjt','project')　　保存 CCS IDE 中的 my.pjt 工程
　　save(cc,[ ],'project')　　保存 CCS IDE 中当前激活的工程
　　save(cc,'all','text')　　保存 CCS IDE 中所有打开的文本文件
　　save(cc,'mysource.c','text')　　保存 CCS IDE 中的 mysource.c 文件
　　save(cc, [ ], 'text')　　保存 CCS IDE 当前窗中激活的文本文件

(61) set 函数：

语法形式：

```
set(cc,'propertyname','propertyvalue')
set(cc,'propname1','propvalue1','propname2','propvalue2')
v =set(cc)
cc.propertyname ='propertyvalue'
set(rx,'propertyname','propertyvalue')
set(rx,'propname1','propvalue1','propname2','propvalue2')
v =set(rx)
rx.propertyname ='propertyvalue'
```

说明：设置对象的属性。propertyname 为属性名。propertyvalue 为修改的属性值。

v =set(cc)和 v =set(rx)返回 CCS IDE 和 RTDX 连接对象的属性及其属性值范围。如果属性值范围不是有限个，则此属性值返回{ }。

```
cc = ccsdsp;
v=set(cc)
v =
  timeout: {}
  page: {}
  eventwaitms: {}
```

设置属性的方法，我们在本章的前面已作了详细介绍，这里不再详述。

(62) symbol 函数：

语法形式：

```
s =symbol(cc)
```

说明：从 CCS IDE 中返回最近加载程序的符号表，并把返回的符号表放入到结构体向量 s 中。s 中的每一元素都是一个结构体，对应于一个符号，结构体包括此符号的名和地址，例如第 i 个元素包括如下成员：

　　s(i).name　　符号名
　　s(i).address(1)　　符号的地址或地址偏移量
　　s(i).address(2)　　符号所在的存储器页，对于不支持存储器页的 DSP，此项为 0。

(63) visible 函数：

语法形式：

visible(cc,state)

说明：设置 CCS IDE 窗口是否在桌面上可见。如果 state=0，则 CCS IDE 窗口在桌面上不可见；如果 state=1，则 CCS IDE 窗口在桌面上可见。尽管 CCS IDE 窗口在桌面上不可见，CCS IDE 仍然可能在后台运行，因而 MATLAB 仍然可以利用 CCS IDE 的功能，但用户不能直接对 CCS IDE 操作。因此如果用户需要直接对 CCS IDE 进行操作，必须利用 visible(cc,1) 使 CCS IDE 在桌面上可见。

(64) write 函数：

语法形式：

```
write(cc,address,data,timeout)
write(cc,address,data)
write(objname)
write(objname,index)
write(objname,stindex,member1,value1,…,membern,valuen,memindex)
write(…,timeout)
```

说明：向 DSP 的存储器中写入数据。

前二种语法形式对连接对象进行写操作，把数据 data 写入到以 address 作为首地址的存储器块内。data 为写入的数据，可以为标量、矢量、二维或多维数据矩阵，支持的数据类型包括：'double', 'int8', 'int16', 'int32', 'single'和'uint8'。write 按列顺序把此数据矩阵写入到 DSP 的存储器中。address 指定写入存储器块的首地址。详见对 isreadable 函数的介绍。

最后四种语法形式对嵌入式对象进行写操作。objname 为嵌入式对象句柄。

index 指定写入数据元素的索引号，index 可以为标量或向量。

- 如果 index 为[ ]或省略，则写入 objname 中的所有数据。
- 如果 index 为标量，则写入 objname 中的第 index 个列矢量。
- 如果 index 为一矢量，则写入 objname 中由 index 指定的元素。例如，index 为[2 2 2]，则写入矩阵中的第[2][2][2]元素值。注意，此时 index 矢量的维数必须与对象的数据维数(size 属性中指定)一致，例如，如果对象为四维矩阵，则 index 也必须为四维矩阵。
- 如果要求写入多个元素，则可以利用 MATLAB 中的标准方法。例如，index 为[1:6] 则写入前 6 个元素(假定对象所访问的符号为一个向量)。

第五种语法形式对嵌入式结构体进行写操作。objname 为嵌入式结构体对象。membern 指定结构体成员名。valuen 为写入 membern 中的值。memberindex 指定成员 membern 中写入元素的索引号。当读取的结构体为 objname 的一个向量元素或结构体成员时，stindex 用来指定此结构体在 objname 中的索引号。

写入的数据类型应与被访问成员的数据类型相一致。write 根据 objname 对象的属性(例如 wordsize,storageunitspervalue,size,represent 和 binarypt 等)来进行数值转换。如果写入的数值超过对象的数据类型所表示的范围时，write 会写入饱和值(此数据类型所表示的最大或最小值)。如果写入的数据元素数目超过指定的存储器块的容量时，write 只取前面的元素放入存储器中而把超过的部分丢掉。如果写入的数据元素数目少于存储器块的容量时，没有写入值的存储器部分不发生改变。如果写入一个字符串，write 会自动在字符串的后面附加一个 NULL 字符。

例　mm 为嵌入式对象，访问嵌入在目标 DSP 中的一个数组变量，维数为 10×1。

write(mm,[1:15])　　只有前 10 个元素值写入到 DSP 的存储器中

write(mm,[1:5])　　存储器数组中的前 5 个值被修改，后 5 个值不变

write(mm,5,6)　　存储器数组中的第 5 个元素值修改为 6

如果 mm 访问一个字符串变量，此字符串变量的当前值为'Hello world'。

write(mm,'ciao')　　写入一个字符串到 DSP 的存储器中，此字符串所在存储空间的值变为'ciao\0 World'。

(65) writemsg 函数：

语法形式：

```
data =writemsg(rx,channelname,data,timeout)
data =writemsg(rx,channelname,data)
```

说明：向指定的 RTDX 通道 channelname 写入数据 data。channelname 指定写通道名，此通道名必须在 DSP 程序中已经定义过。data 为写入的数据矩阵，此矩阵中的所有元素必须具有相同的数据类型。它支持的数据类型包括：uint8、int16、int32、single 和 double。writemsg 按列顺序把此数据矩阵写入到 RTDX 通道中。

5.5　CCSLink 演示例子

本节演示如何应用 CCSLink 进行目标程序调试、测试的整个过程，详细介绍了 CCS IDE 连接对象、RTDX 连接对象和嵌入式对象的使用方法和操作步骤。

通过本节的学习，读者就可以应用 CCSLink 调试自己的嵌入式目标程序了。

5.5.1　CCS IDE 连接对象应用演示

CCS IDE 连接对象提供 MATLAB 与 CCS IDE 和目标 DSP 的连接，利用此连接可以在 MATLAB 下控制和操作 DSP 中的应用程序，利用 MATLAB 中强大的计算、分析和可视化工具来分析和对比目标程序运行过程中的结果，大大缩短嵌入式应用程序的开发调试周期。

本节假定用户已经安装并配置好 CCSLink 和 CCS IDE，而且要求在主机(即计算机)系统上已经安装好一块目标板或软件模拟器，例如，DSK 或 EVM。本节的演示过程包括：

(1) 选择目标 DSP。

(2) 创建 CCS IDE 连接对象。

(3) 利用 MATLAB 把文件加载到 CCS IDE 中。

(4) 在 MATLAB 环境下对 CCS IDE 连接对象进行操作。

(5) 关闭 CCS IDE 连接对象。

1. 选择目标 DSP

(1) 利用 ccsboardinfo 函数列出安装在主机上的目标板及其 DSP 信息，显示如下：

```
ccsboardinfo
  Board   Board                              Proc   Processor                 Processor
  Num     Name                               Num    Name                      Type
  .....  ................................  ......  ......................................................
  1   C6xxx Simulator (Texas Instrum)…   0      6701                      TMS320C6701
  0   C6x11 DSK (Tecas Instruments)      0      CPU                       TMS320C6x1x
```

(2) 选择 C6x11 目标板上的 0 号 DSP：

```
boardnum=0;  procnum=0;
```

2. 创建 CCS IDE 连接对象

(1) 利用 ccsdsp 函数创建一个连接对象：

```
cc=ccsdsp('boardnum', boardnum, 'procnum', procnum)
```

cc 为连接对象句柄。利用 ccsdsp 创建连接对象时，MATLAB 会自动启动 CCS IDE，CCS IDE 在后台开始运行。

(2) 使 CCS IDE 窗口在桌面上可见：

```
visible(cc,1)
```

即使 CCS IDE 在后台运行，MATLAB 也可以利用 CCS IDE 的功能。但用户在开发过程中经常需要直接对 CCS IDE 进行操作，因此需要 CCS IDE 窗口出现在桌面上。

(3) 利用 info、disp、isrunning 或 isrtdxcapable 等函数来测试目标板和 DSP 的状态信息：

```
linkinfo =info(cc)
linkinfo =
        procname: 'CPU'
      isbigendian:0
           family:320
        subfamily:103
        revfamily:1
          timeout:10
runstatus =isrunning(cc)
runstatus =
          0
usesrtdx =isrtdxcapable(cc)
usesrtdx =
          1
```

3. 利用 MATLAB 把文件加载到 CCS IDE 中

(1) 把工程文件加载到 CCS IDE 中：

```
projfile =fullfile(MATLABroot, 'toolbox', 'tiddk', 'tidemos',…)
'ccstutorial', 'ccstut_6x11.pjt');
projpath =fileparts(projfile);
open(cc, projfile)               % 加载工程文件
```

ccstut_6x11.pjt 是 CCSLink 提供的一个工程文件例子。

(2) 编译链接 CCS IDE 中当前的工程文件，生成目标 DSP 可执行文件:

```
build(cc)
```

(3) 把可执行文件加载到目标 DSP 中:

```
load(cc, 'ccstut_6x11.out')
```

4. 在 MATLAB 环境下对 CCS IDE 连接对象进行操作

在 CCS IDE 的工程视窗中可以看到一个源文件 ccstut.c，此源文件中具有两个全局数组：ddat 和 idat。

```
int16_T   idat[] = { 1,508,647,7000};
double   ddat[] = { 16.3, - 2.13, 5.1, 11.8 };
```

下面演示利用 CCSLink 中的 read 和 write 函数来访问这两个全局数组。

(1) 插入一个断点(breakpoint)。若执行

```
insert(cc, 'ccstut.c', 64, 'break');
```

则在 ccstut.c 文件的第 64 行处插入一个断点。

(2) 控制目标 DSP 的执行：

```
halt(cc)                      % 停止目标DSP
restart(cc)                   % 复位程序计数器PC指向程序的入口处
run(cc, 'runtohalt', 30);     % 运行程序，直到断点处才把控制权返回MATLAB
```

(3) 访问嵌入在目标 DSP 程序中的符号。若执行

```
ddatv =read(cc,address(cc,'ddat'),'double',4)
ddatv =
     16.3000    - 2.1300    5.1000    11.8000
```

则从目标存储器的 ddat 符号处，读入 4 个双精度浮点数，并返回到 MATLAB 的 ddatv 中。

若执行

```
idatv =read(cc,address(cc,'idat'),'int16',4)
idatv =
     1       508       647       7000
```

则从目标存储器的 idat 符号处，读入 4 个 int32 数值，并返回到 MATLAB 的 idatv 中。

若执行

```
write(cc,address(cc,'ddat'),double([pi   12.3   exp(- 1)   sin(pi/4)]))
```

则修改目标存储器 ddat 中的数值。

若执行

```
ddatv =read(cc,address(cc,'ddat'),'double',4)
ddatv =
     3.1416      12.3000      0.3679       0.7071
```

则读 ddat 修改后的数值。

(4) 访问目标 DSP 中的寄存器。若执行

```
reg1=regread(cc, 'A0', '2scomp');
```

则读 A0 寄存器中的值，并根据 A0 中数值的格式(这里是二进制补码符号整数)将其转换成双精度数，分配给 MATLAB 中的 reg1。

若执行

```
reg2=cc,regread('B2', 'binary');
```

则读 B2 寄存器中的值，并根据 B2 中数值的格式(无符号二进制整数)将其转换成双精度数，分配给 MATLAB 中的 reg2。

若执行

```
cc, regwrite('A2', reg1, '2scomp');
```

则把 reg1 中的数值转换成二进制补码格式写入到 A2 寄存器中。

5. 关闭 CCS IDE 连接对象

利用 clear 函数删除前面创建的 CCS IDE 连接对象句柄 cc：

```
clear cc
```

5.5.2　嵌入式对象应用演示

利用 MATLAB 中的面向对象编程技术和 CCSLink，可以为目标程序中的所有 C 符号创建嵌入式对象来代表和操作此符号。

本节演示嵌入式对象的操作方法，要求用户已经按上节的操作过程创建了一个 CCS IDE 连接对象，并把可执行文件 ccstut_6x11.out 加载到目标 DSP 中。

1. 创建嵌入式对象

(1) 复位程序计数器指向程序的入口处:

```
restart(cc)
```

(2) 把程序计数器定位到 main(主)函数的开始处:

```
goto(cc, 'main')
```

此操作确保 C 全局变量被初始化。

(3) 创建一个嵌入式对象:

```
cvar=createobj(cc, 'idat')
    NUMERIC Object
      Symbol Name                 : idat
      Address                     : [ 40060 0 ]
      Wordsize                    : 16 bits
      Address Units per value     : 2 AU
      Representation              : signed
      Binary point position       : 0
      Size                        : [ 4 ]
      Total address units         : 8 AU
      Array ordering              : row - major
      Endianness                  : little
```

上例创建一个嵌入式对象 cvar 来代表符号表中的全局数组 idat。利用嵌入式对象 cvar 就可以在 MATLAB 空间中对数组 idat 进行访问和操作。

2. 访问嵌入式变量

如有以下例子：

```
read(cvar)
    ans =
      1     508     647     7000
```

则读嵌入式数组 idat 中的所有元素值。再如

```
read(cvar,2)
  ans=
      508
```

则读数值 idat 中的第二个元素值。再如

```
write(cvar,4,7001)
```

则修改数组 idat 中的第四个元素值。再如

```
read(cvar,size(cvar))
   ans=
       7001
```

则读数组 idat 中的第四个元素值。

3. 对嵌入式对象进行操作

利用 cast、convert 和 size 等函数来修改 MATLAB 中的嵌入式对象 cvar，但它们对 DSP 中的数组 idat 不会产生影响。如有语句

```
set(cvar, 'size', [2]);
```

则设置 cvar 对象的 size 属性值，使其变为前两个元素值。再如

```
read(cvar)
   ans=
       1     508
```

则只返回前两个元素值。再如

```
uintcvar=cast(cvar, 'unsigned short');
```

则复制 cvar 到一个新对象 uintcvar 中，但对象 uintcvar 的 represent 属性值由 cvar 的 signed short 变为 unsigned short。uintcvar 会按无符号整数来解释 idat 中的每一个元素值。

```
convert(cvar, 'unsigned short')
     ans=
         Symbol Name                  : idat
         Address                      : [ 40060 0 ]
         Wordsize :16 bits
         Address Units per value       : 2 AU
         Representation               : unsigned
         Binary point position        : 0
         Size                         : [ 2 ]
         Total address units          : 4 AU
         Array ordering               : row - major
         Endianness                   : little
```

上例改变对象 cvar 的 represent 属性值为 unsigned short。

4. 对其它数据类型符号进行访问

工程源文件 ccstut.c 中定义了一个结构体变量和一个字符串变量：

```
struct TAG_myStruct {
    int iy [2 ][3 ];
    myEnum iz;
```

```
    }myStruct ={{{1,2,3},{4, - 5,6}},MATLABLink}
char myString[] = "Treat me like an ANSI String";

cvar =createobj(cc,'myStruct')
    cvar=
        Symbol Name            : myStruct
        Address                : [ 40032 0 ]
        Size                   : [ 1 ]
        Total Address Units    : 28 AU
        Members                : 'iy','iz'
```

上例创建了一个结构体对象 cvar 来代表和访问嵌入在 DSP 中的结构体 myStruct。又如

```
read(cvar)
    ans =
        iy:[2x3 double ]
        iz:'MATLABLink'
```

则读结构体 myStruct 的成员及其值。再如

```
write(cvar,'iz','Simulink');
```

则修改结构体 myStruct 中的 iz 成员值，iz 为枚举类型。又如

```
cfield =getmember(cvar,'iz')
  ENUM Object
    Symbol Name                  : iz
    Address                      : [ 40056 0 ]
    Wordsize                     : 32 bits
    Address Units per value      : 4 AU
    Representation               : signed
    Binary point position        : 0
    Size                         : [ 1 ]
    Total address units          : 4 AU
    Array ordering               : row-major
    Endianness                   : little
    Labels &values               : MATLAB=0,Simulink=1,SignalToolbox=2,
                                   MATLABLink=3,EmbeddedTargetC6x=4
```

则创建一个新的枚举类型对象 cfield 来代表和直接访问结构体 myStruct 中的成员 iz。如有

```
write(cfield,4);
```

则利用对象 cfield 可直接对结构体成员 iz 进行修改。如有

```
cstring =createobj(cc,'myString');
```

则创建一个字符串对象 cstring 来代表和访问嵌入在 DSP 中的字符串变量 myString。如有

```
read(cstring)
  ans=
      Treat me like an ANSI String
```

则读字符串变量 myString 中的值。如有

```
write(cstring,7, 'ME');
```

则修改字符串变量 myString 中的第 7、第 8 个字符。如有

```
read(cstring)
    ans=
        Treat ME like an ANSI String
```

则读修改后的字符串变量 myString。如有

```
write(cstring,1,127);
```

则修改字符串变量 myString 中的第 1 个字符，使其变为 ASCII 值 127 所代表的字符。如有

```
readnumeric(cstring)
    ans=
        127   114   101   97   116   32   77   69   32   108   105   107
        101   32    97    110   32   65   78   83   73    32    83   116
        114   105   110   103   0
```

则查看 myString 中的 ASCII 值。

5. 清除创建的嵌入式对象

用 clear 可清除创建的嵌入式对象。如

```
clear   cvar   uintcvar   cfield
```

则清除前面创建的嵌入式对象 cvar、uintcvar 和 cfield。

5.5.3　RTDX 连接对象应用演示

利用 RTDX 连接对象可以允许 MATLAB 与 DSP 程序之间实时地交换信息，而不必停止目标 DSP 上正在执行的程序。利用这种实时交互式功能，开发者可以更加容易地看清程序的运行过程、状态和中间运行结果。

本节演示过程包括如下四部分：

(1) 创建 RTDX 连接对象。

(2) 配置 RTDX 通道。

(3) 在 MATLAB 环境下对 RTDX 连接对象进行操作。

(4) 清除 RTDX 连接对象。

本节假定加载的目标 DSP 执行程序中有如下一段程序代码：

```
 RTDX_CreateInputChannel(ichan);   /*声明 ichan 输入通道*/
 RTDX_CreateOutputChannel(ochan); /*声明 ochan 输出通道*/
 short recvd[10];
 while (!RTDX_isInputEnabled(&ichan))
 {/*等待 MATLAB 使能输入通道*/}
 RTDX_read(&ichan,recvd,sizeof(recvd));/*读输入通道中数据，放入到 recvd 数组中*/
 puts("\n \n Read Completed ");
 for (j=1;j<=20;j++){
    for (i=0;i<10; i++){
        recvd [i ] ++=1;
        }
while (!RTDX_isOutputEnabled(&ochan))
      {/*等待 MATLAB 使能输出通道*/}
RTDX_write(&ochan,recvd,sizeof(recvd));/*向输出通道写入处理结果，处理结果在 recvd 数组中*/
while (RTDX_writing !=NULL )
      {/*等待中断完成数据传送*/}
 }
```

1. 创建 RTDX 连接对象

RTDX 连接对象是 CCS IDE 连接对象的一个成员，因此在创建 CCS IDE 连接对象的同时也创建了 RTDX 连接对象。

(1) 利用 ccsboardinfo 显示安装的目标板和 DSP 信息：

```
ccsboardinfo
```

(2) 创建连接对象：

```
cc=ccsdsp('boardnum', 0,'procnum', 0);
```

(3) 加载目标 DSP 的可执行文件：

```
load(cc, 'myRTDX.out');
```

2. 配置 RTDX 通道

在利用 RTDX 通道进行数据传递之前，必须打开并且使能这些 RTDX 通道。打开的 RTDX 通道名必须在目标 DSP 的应用程序中已被定义和声明，否则不能打开。也就是说，不能在 CCSLink 中随便打开一个通道，这个通道名必须在目标 DSP 的程序中已被定义，详见第 3 章的 CCS 介绍。

(1) 配置 RTDX 通道缓冲器：

```
cc.rtdx.configure(1024,4);    %定义 4 个通道缓冲器，每个通道 1024 个字节
```

通道缓冲器是可选的。目标 DSP 发送数据的速度一般超过 MATLAB 读数据的速度，因此添加缓冲器后，确保不会丢失数据。

(2) 打开并使能一个 RTDX 写通道：

```
cc.rtdx.open('ichan', 'w');    %打开名为'ichan'的 RTDX 写通道(MATLAB 向 DSP 写)
cc.rtdx.enable('ichan');       %使能 RTDX 写通道'ichan'
```

(3) 打开并使能一个 RTDX 读通道：

```
cc.rtdx.open('ochan','r');     %打开名为'ochan'的 RTDX 读通道(MATLAB 从 DSP 读)
cc.rtdx.enable('ochan');       %使能 RTDX 读通道'ochan'
```

(4) 使能 RTDX 接口：

```
cc.rtdx.enable;
```

在使用 RTDX 通道之前，也必须使能 RTDX 接口，使能 RTDX 接口与使能单个 RTDX 通道的顺序可前可后。

(5) 查看 RTDX 对象属性：

```
cc.rtdx
  RTDX Object:
      API version:       1.0
      Default timeout:   20.00 secs
      Open channels:     2
```

可以看到当前已打开 2 个 RTDX 通道。

3. 对 RTDX 连接对象进行操作

利用 RTDX 通道可以把数据发送给目标 DSP 进行处理，并把处理结果再经过 RTDX 通道读入 MATLAB 中进行分析，所有这些操作都可以在 MATLAB 环境下完成。

(1) 复位程序计数器(PC)，使其指向 DSP 程序的入口处：

```
restart(cc);
```

(2) 运行 DSP 应用程序：

```
run(cc, 'run');
```

(3) 验证写通道是否已使能：

```
cc.rtdx.isenabled('ichan')
        ans=
            1
```

如果 ans=0 表明通道没有使能，因此必须重新使能和验证，直到 isenabled 返回 ans=1。

(4) 向 RTDX 通道写入数据：

```
cc.rtdx.iswritable('ichan')                    %检查 ichan 通道是否可写
   ans=
        1
cc.rtdx.writemsg('ichan', int16([1:10]);       %向 ichan 通道写入数据
```

(5) 验证读通道是否已使能：

```
cc.rtdx.isenabled('ochan')
      ans=
          0
```

ans=0 表明输入通道 ochan 没有使能，因此必须重新使能并验证，直到 ans=1，即

```
cc.rtdx.enable('ochan');
cc.rtdx.isenabled('ochan')
        ans=
            1
```

(6) 检查输出通道中的信息数目：

```
num_of_msgs=cc.rtdx. msgcount ('ochan')
    num_of_msgs=
                 20
```

可以利用一段循环体程序连续检查 ochan 通道中的信息数目，直到信息数目为 20。

(7) 从 ochan 通道中读入一个信息：

```
outdata=cc.rtdx.readmsg('ochan', 'int16')
      outdata=
              2    3    4    5    6    7    8    9    10    11
```

(8) 把信息读入一个单元阵中：

```
outdata=rtdx.readmsg('ochan', 'int16', 3)
outdata=
        [1× 10 int16]    [1× 10 int16]    [1× 10 int16]
```

从 ochan 通道读入 3 个信息放入 outdata 单元阵中，每一信息都为 1×10 矢量：

```
outdata{1,1}=3   4   5   6   7   8   9   10   11   12
outdata{1,2}=4   5   6   7   8   9   10   11   12   13
outdata{1,3}=5   6   7   8   9   10  11   12   13   14
```

(9) 读入两条信息到两个 2×5 矩阵中：

```
outdata=cc.rtdx, readmsg('ochan', 'int16', [2 5],2)
   outdata=
      [2× 5 int16]   [2× 5 int16]
     outdata={1,:}
```

```
ans=
     6    8    10    12    14
     7    9    11    13    15
ans=
     7    9    11    13    15
     8    10   12    14    16
```

(10) 利用 readmat 函数读入两个信息到 4×5 矩阵中：

```
outdata =cc.rtdx.readmat('ochan', 'int16', [4 5])
   outdata =
        8    12   16   11   15
        9    13   19   12   16
        10   14   9    13   17
        11   15   10   14   18
```

(11) 查看 ochan 通道中余下的信息数目：

```
num_of_msgs =cc.rtdx.msgcount('ochan')
   num_of_msgs =
            12
```

前面的操作过程已经读走了 8 条信息，因此当前 ochan 通道中只剩下 12 条信息。

(12) 利用 flush 函数删除 ochan 通道中的 1 条信息：

```
cc.rtdx.flush('ochan', 1)
num_of_msgs =cc.rtdx.msgcount('ochan')
 num_of_msgs =
          11
```

(13) 清空 ochan 通道中所有余下的信息：

```
cc.rtdx.flush('ochan', 'all')
num_of_msgs=cc.rtdx.msgcount('ochan')
   num_of_msgs=
            0
```

4．清除 RTDX 通道

利用 RTDX 完成任务后，最后都必须清除掉所有不需要的 RTDX 通道，以免对接下来的程序调试带来影响。经过下面的步骤可关闭所有的 MATLAB 和指定 DSP 的连接。

(1) 停止目标 DSP：

```
if(isrunning)      %检查 DSP 是否已停止
 cc.halt;          %停止 DSP
end
```

停止 DSP 之前，最好先检查一下 DSP 是否已停止，以免对 DSP 带来损害。

(2) 禁止 RTDX 通道和 RTDX 接口：

```
cc.rtdx.disable('all');
```

禁止 cc 中所有打开的 RTDX 通道。

```
disable(cc.rtdx);
```

禁止 cc 中的 RTDX 接口。

(3) 关闭 RTDX 通道：

```
close(cc.rtdx, 'all');
```

(4) 清除 MATLAB 与指定目标的所有连接：

```
clear(cc)
```

如果 CCSLink 中创建了与多个 DSP 的连接对象，上述过程只清除了由 cc 指定的 DSP 的连接关系，而与其它 DSP 的连接关系不受影响。

思　考　题

5.1　MATLAB Link for CCS Development Tools 工具有何功能？MATLAB 从何版本开始提供此工具？MATLAB Link for CCS Development Tools 工具可以支持 TI 公司的哪些类型的 DSP 及开发板？是否也支持用户自己开发的 DSP 板？

5.2　MATLAB、MATLAB Link for CCS Development Tools、CCS 和目标 DSP 板之间的关系如何？

5.3　利用 MATLAB Link for CCS Development Tools 工具，能否只在 MATLAB 环境下就可以完成 DSP 代码生成、调试的整个过程，即无须再切换到 CCS 下？

5.4　何为面向对象编程？何为对象属性？MATLAB Link for CCS Development Tools 工具提供的对象有哪些？什么是连接对象？什么是嵌入式对象？它们各有什么功能？它们之间有何内在联系？

5.5　MATLAB Link for CCS Development Tools 工具提供的函数有哪些？(在 MATLAB 环境下就是利用这些函数来对 CCS 和目标 DSP 进行访问的。)

5.6　利用 MATLAB Link for CCS Development Tools 工具能否对多 DSP 系统同时调试？如何进行？

5.7　编写一段对输入数据进行 FIR 滤波的 C 程序，生成可执行代码并加载到目标 DSP 中，然后在 MATLAB 中创建一个滤波器系数的嵌入式对象，利用 MATLAB 中的 FDATool 工具设计滤波器系数，并把此系数通过嵌入式对象直接输出到目标 DSP 中。

5.8　创建一个 RTDX 对象，利用 RTDX 通道向目标 DSP 中输出上题中的滤波器系数，并比较上述两种方法的优缺点。

第 6 章　由 Simulink 模型生成 TI C6000 DSP 的目标代码

第 5 章介绍的 MATLAB Link for CCS Development Tools 工具，为 TI DSP 实时应用开发的调试和测试阶段提供了强大的支持，而本章介绍的 Embedded Target for the TI TMS320C6000™ DSP Platform 产品更有意义，它为 TI C6000 DSP 实时应用开发的整个过程(概念设计、算法仿真、源代码编写、目标代码生成、调试和测试)都提供了支持。

Embedded Target for the TI TMS320C6000™ DSP Platform 也集成在 MATLAB6.5(R13)中，在此产品中 MathWorks 公司与 TI 公司集成了先进的 DSP 软件技术。利用 Embedded Target for the TI TMS320C6000™ DSP Platform 能够从 Simulink 模型自动生成 TI C6000 DSP 的可执行目标代码，并且为 TI C6701 EVM 和 TI C6711 DSK 目标板上的 I/O 设备提供驱动代码。在 Simulink 模型中加入 TI C6701 EVM 或 TI C6711 DSK 板支持模块后，Simulink 模型就可以直接在 TI C6701 EVM 或 TI C6711 DSK 板上进行实时测试，从而在 Simulink 统一环境下，就可以实现整个硬件在线仿真。

本章的内容包括：

6.1 节是 Embedded Target for the TI TMS320C6000™ DSP Platform 概述，它为读者提供 Embedded Target for the TI TMS320C6000™ DSP Platform 的内容概述，包括功能特点、配置方法和实时代码的开发步骤等。

6.2 节介绍如何设置 Real－Time Workshop 编译链接选项。Real－Time Workshop 先从 Simulink 模型生成 C 源代码，然后再调用 TI 的编译器、汇编器和链接器等开发工具来生成目标可执行代码，Real－Time Workshop 面板中的选项控制着这一过程的操作。设置 Real－Time Workshop 选项是代码开发的必要步骤，本节详细介绍如何设置每一选项。

6.3 节介绍如何在生成的可执行代码中集成 DSP/BIOS 功能。

6.4 节介绍如何把利用 FDATool 设计的滤波器系数输出到目标 DSP 中。FDATool 是信号处理工具箱中的一种可视化滤波器分析设计工具。MATLAB Link for CCS Development Tools 把 FDATool 和 CCS 连接在一起，在 FDATool 中设计好的滤波器系数通过 CCS 可直接输出到目标 DSP 的存储器中，实时测试滤波器的性能。本节详细介绍如何从 FDATool 工具向目标 DSP 的存储器中输出滤波器系数。

6.5 节是 C6000lib 模块库，介绍 Embedded Target for the TI TMS320C6000™ DSP Platform 提供的 Simulink 模块，包括各个模块的功能和使用方法等。

6.6 节是由 Simulink 模型生成实时代码的过程，它对整个目标代码生成过程进行总结，包括注意的事项等。

6.7 节是 TI C6701 EVM 目标板的应用，针对应用 TI C6701 EVM 目标板的用户，详细介绍如何为 TI C6701 EVM 目标板开发实时模型，并利用 Embedded Target for the TI TMS320C6000™ DSP Platform 提供的例子演示详细的开发过程。

6.8 节是 TI C6711 DSK 目标板的应用，针对应用 TI C6711 DSK 目标板的用户，详细介绍如何为 TI C6711 DSK 目标板开发实时模型，并利用 Embedded Target for the TI TMS320C6000™ DSP Platform 提供的例子演示详细的开发过程。

为了表述简捷，本章把 Embedded Target for the TI TMS320C6000™ DSP Platform 简称为 ETTIC6000。

6.1 ETTIC6000 概述

6.1.1 ETTIC6000 的功能和特点

ETTIC6000 利用 Real－Time Workshop 直接从 Simulink 模型生成 TI C6000 DSP (C67x 浮点和 C62x/C64x 定点 DSP)的高效代码，不再需要传统的 DSP 编程过程。ETTIC6000 能够自动生成 CCS 工程，并且为 TI 的 C6701 EVM 和 C6711 DSK 目标板提供支持。开发人员在 Simulink 环境下利用 DSP Blockset、Fixed－Point Blockset 和 ETTIC6000 提供的 C62x 汇编语言函数模块，构造系统模型和实时 DSP 算法，并进行模型仿真，一旦仿真结果满意，就可以插入 ETTIC6000 提供的 TI C6701 EVM 或 TI C6711 DSK 目标板上的 I/O 模块，ETTIC6000 能够自动完成代码产生、代码加载、执行及与目标 DSP 进行通信等功能。利用这种从概念设计到实时实现的集成开发环境，可以大大缩短产品开发周期，加快产品的上市时间。

ETTIC6000 首先利用 Real－Time Workshop 从 Simulink 模型生成标准 ANSI C 程序代码，然后通过 MATLAB Link for CCS Development Tools 调用 CCS 开发工具，编译链接这些 C 代码，生成指定目标板(C6701 EVM、C6711 DSK 或 C6xxx Simulator)的可执行代码，并把生成的可执行代码加载到目标板中进行算法实时性评估。ETTIC6000 利用 MATLAB Link for CCS Development Tools 对 DSP 实时应用程序进行交互式调试和测试。

Embedded Target for TI C6000 DSP、Simulink、Real－Time Workshop、CCS 和 TI 目标板之间的关系如图 6.1 所示。

ETTIC6000 的主要特点包括：

- 由 Simulink 模型自动生成 TI C6000 定点和浮点 DSP 的可执行代码。
- 能够在 C6711 DSK 和 C6701 EVM 目标板上进行算法实时性评估。
- 提供优化的 C62x DSP 汇编语言函数模块库。
- 支持 RTDX，提供 Simulink 和目标 DSP 程序之间双向、实时数据传递。
- 支持 DSP/BIOS 实时操作系统，能够实时分析和优化代码。
- 符合 eXpress DSP 插件标准。
- 在 Simulink 统一环境下，完成从概念设计到实时实现的整个过程。

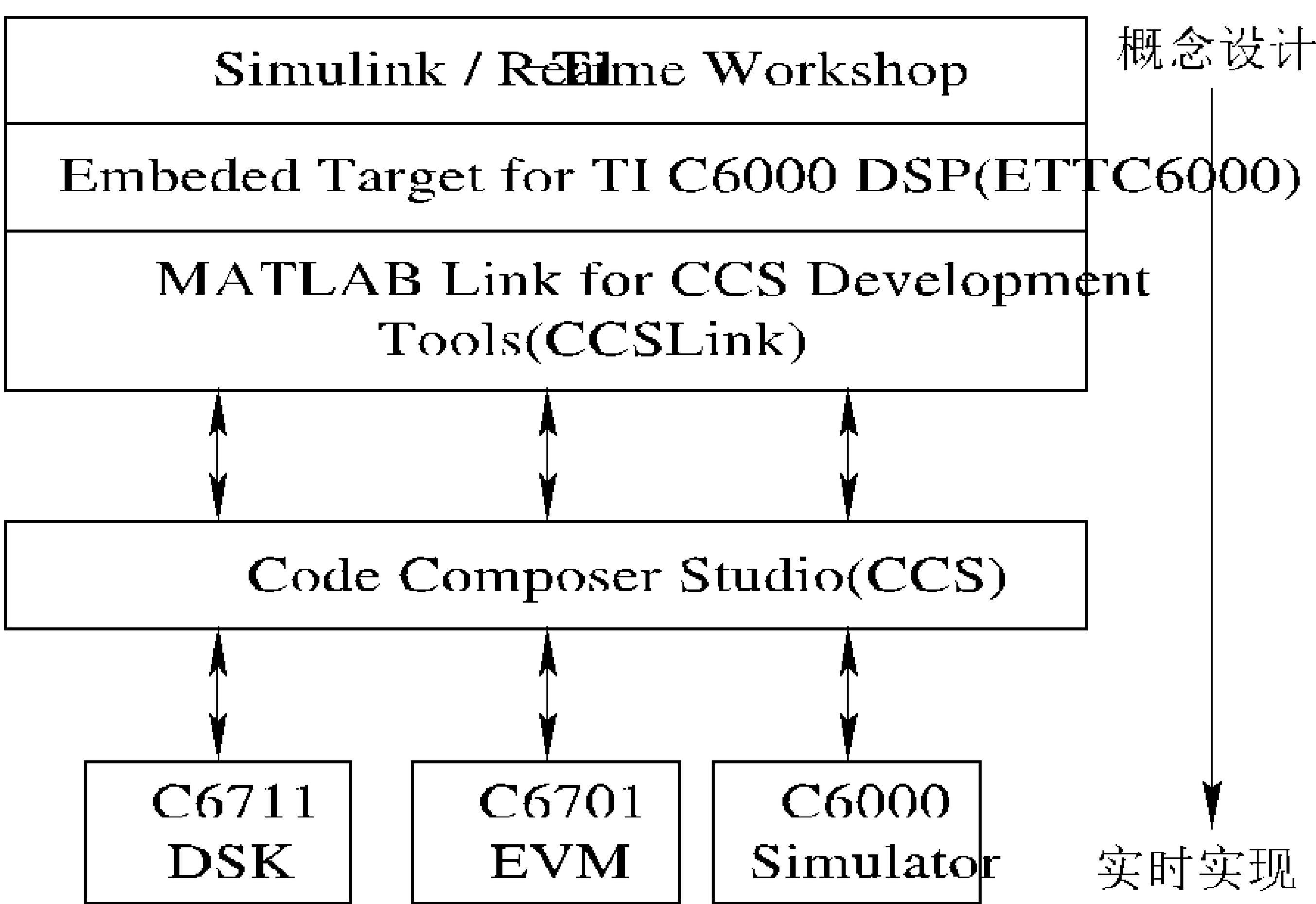


图 6.1 Embedded Target for TI C6000 DSP 和其它产品之间的关系

6.1.2 ETTIC6000 的配置

ETTIC6000 必须依靠 MathWorks 公司和 TI 公司产品的支持才能够正常工作，因此利用 ETTIC6000 进行嵌入式实时系统开发时，必须安装其需要的 MathWorks 公司和 TI 公司的产品。

MathWorks 公司产品包括：

MATLAB6.1、Signal Processing Toolbox5.0、Simulink、DSP Blockset、Fixed－Point Blockset(应用于定点 DSP)、Real－Time Workshop3.0 和 MATLAB Link for CCS Development Tools。

TI 公司产品包括：

软件：CCS 开发工具(包括汇编器、编译器、链接器、CCS IDE、CCS 配置程序及其它工具)。

目标平台：TMS320C6701 EVM(C6701 评估板)、TMS320C6711 DSK(C6711 学习板)、C6xxx Simulator(CCS 中的软件模拟器)。

事实上，无论是 TI 公司还是第三方或用户自己开发的 C6000™ 目标板，只要支持 JTAG 和 RTDX 通信，就都可以利用 ETTIC6000 开发嵌入式实时应用系统。

安装完 ETTIC6000 及其所需的产品后，需要验证是否安装成功，其方法是：在 MATLAB 命令窗中输入如下命令：

```
c6000lib
```

MATLAB 会显示一个 Library:C6000lib 窗口，此窗口中包含 ETTIC6000 提供的如下 4 种模块库：

- C6711 DSK Board Support;
- C6701 EVM Board Support;
- C62x DSP Library;
- RTDX Instrumentation。

如果 MATLAB 没有显示上述窗口或 MATLAB 不能识别 c6000lib 命令，就需要重新安装 ETTIC6000。

要验证 CCS 是否安装成功，需在 MATLAB 命令窗中输入如下命令：

ccsboardinfo

如果 CCS 安装并配置好，MATLAB 会返回类似于下列形式的目标板和 DSP 信息：

Board Num	Board Name	proc Num	processor Name	processo Type
……	……………………………………	……	……………………………………………………	
1	C6xxx Simulator (Texas Instrum …	0	6701	TMS320C6701
0	C6x11 DSK (Texas Instruments)	0	CPU	TMS320C6x1x

如果 MATLAB 没有返回任何目标板信息，则需要重新安装或配置 CCS。

最后还需要测试一下 CCS IDE 是否能够启动。ETTIC6000 会自动启动 CCS IDE，进行编译链接并加载可执行代码。

6.1.3　ETTIC6000 的模块库

ETTIC6000 提供了多个 Simulink 模块，以支持 C6711 DSK 板、C6701 EVM 板、RTDX 实时数据传递和 C62x 系列定点 DSP，这些模块分别位于 4 种模块库中，如图 6.2 所示。在 MATLAB 中利用命令 C6000lib 就可以看到这些模块库及模块。本节列出这些模块及其功能，详细的使用方法和参数设置在 6.5 节中再介绍。

图 6.2　ETTIC6000 提供的 Simulink 模块

(1) C6701 EVM 板支持模块库：

- C6701 EVM ADC：配置 C6701 EVM 板上的 A/D 转换器。
- C6701 EVM DAC：配置 C6701 EVM 板上的 D/A 转换器。
- C6701 EVM DIP Switch：模拟或直接读 C6701 EVM 板上的 DIP 开关设置。
- C6701 EVM LED：控制 C6701 EVM 板上的发光二极管(LED)。
- C6701 EVM Reset：复位当前的 C6701 EVM 板。

(2) C6701 DSK 板支持模块库：

- C6711 DSK ADC：配置 C6711 DSK 板上的 A/D 转换器。
- C6711 DSK ADC：配置 C6711 DSK 板上的 D/A 转换器。
- C6711 DSK DIP Switch：模拟或直接读 C6711 DSK 板上的 DIP 开关设置。
- C6711 EVM LED：同时控制 C6711 DSK 板上的三个发光二极管(LED)。
- C6711 EVM Reset：复位当前的 C6711 DSK 板。

(3) RTDX 模块库：

- From RTDX：添加一个 RTDX 输入通道，利用此通道可以从 MATLAB 向目标 DSP 程序发送数据。
- To RTDX：添加一个 RTDX 输出通道，利用此通道，目标 DSP 程序可以向 MATLAB 输出数据。

(4) C62x DSPlib 模块库：

- Convert Floating－Point to Q.15：把一个浮点信号转换成 Q15 格式的定点信号。
- Convert Q.15 to Floating－Point：把一个 Q15 格式的定点信号转换成单精度浮点信号。
- Complex FIR：利用复数 FIR 滤波器对输入复数信号进行滤波。
- General Real FIR：利用实 FIR 滤波器对输入实信号进行滤波。
- LMS Adaptive FIR：利用自适应最小均方 FIR 滤波器对输入实信号进行滤波。
- Radix－4 Real FIR：利用基 4 实 FIR 滤波器对输入实信号进行滤波。
- Radix－8 Real FIR：利用基 8 实 FIR 滤波器对输入实信号进行滤波。
- Real Forward Lattice ALL－Pole IIR：利用实前向格形 IIR 滤波器对输入实信号进行滤波。
- Real IIR：利用实 IIR 滤波器对输入实信号进行滤波。
- Symmetric Real FIR：利用对称实 FIR 滤波器对输入实信号进行滤波。
- Autocorrelation：计算输入实序列或帧基矩阵的自相关。
- Block Exponent：返回输入信号每一通道中所有元素的最小冗余符号位数。
- Matrix Multiply：计算两个输入实矩阵的乘积。
- Matrix Transpose：计算矩阵转置。
- Reciprocal：计算一个输入实信号倒数的小数和指数部分。
- Vector Dot Product：计算两个输入实信号的矢量的内积。
- Vector Maximum Index：计算输入实信号的每一通道中最大元素的位置。
- Vector Maximum Value：计算输入实信号的每一通道中元素的最大值。
- Vector Minimum Value：计算输入实信号的每一通道中元素的最小值。
- Vector Multiply：计算两个输入实信号的矢量点乘(对应元素相乘)。

- Vector Negate：对输入实信号或复信号的每一元素值取反。
- Vector Sum of Squares：计算输入实信号的每一通道元素的平方和。
- Weighted Vector Sum：计算两个输入实信号的加权和。
- Bit Reverse：对输入复数信号每一通道中元素的位置进行位反转。
- FFT：对输入复数信号作 FFT。
- Radix－2 FFT：对输入复数信号作基 2 FFT。
- Radix－2 IFFT：对输入复数信号作基 2 IFFT。

6.1.4　应用 ETTIC6000 开发实时 DSP 处理的过程

应用 ETTIC6000 开发实时 DSP 处理的过程，一般经过如下几步：

(1) 概念构思和 DSP 处理算法设计。

(2) 在 Simulink 环境下，利用 DSP Blockset 、Fixed－Point Blockset、C62x DSPlib 和 Simulink 等库中的模块构建算法模型，并在 Simulink 环境下进行仿真。

(3) 如果 Simulink 仿真结果满意，就可以在模型中加入需要的 C6701 EVM 或 C6711 DSK 目标板上的 I/O 模块。

(4) 设置 Real－Time Workshop 中的编译链接(Build)选项。

Real－Time Workshop 能够从 Simulink 模型中自动产生 C 代码并且插入 ADC 和 DAC 模块指定的 I/O 设备驱动程序。这些 I/O 设备驱动程序作为内嵌 S 函数(详见 Simulink 用户手册)插入在产生的 C 代码中。

必须指定正确的系统目标文件(system target file)和 makefile 模板文件(template makefile)文件。对于 C6701 EVM 或 C6711 DSK 目标板，指定 ti_c6000.tlc 作为系统目标文件。

设置好 Build 选项后，通过点击 Real－Time Workshop 面板上的 Build 按钮，Real－Time Workshop 会自动产生实时可执行代码，如果 Build action 中选择 Build and execute，则可执行代码会自动加载到指定的目标板上并且开始运行。

(5) 实时代码调试。

利用 CCS 中的调试工具、MATLAB Link for CCS Development Tools 或 RTDX 来调试目标 DSP 中的程序。

6.2　设置 Real－Time Workshop 编译链接选项

Real－Time Workshop 编译链接(Build)选项用来控制实时代码生成过程，因此利用 ETTIC6000 从 Simulink 模型生成实时代码时，必须先设置好 Real－Time Workshop 中的 Build 选项。

打开 Real－Time Workshop Build 选项设置面板：Simulink 模型窗口中的 Simulation 菜单 → Simulation Parameters 对话框 → Real－Time Workshop 面板。图 6.3 为打开的 Simulation Parameters 对话框及其 Real－Time Workshop 面板。

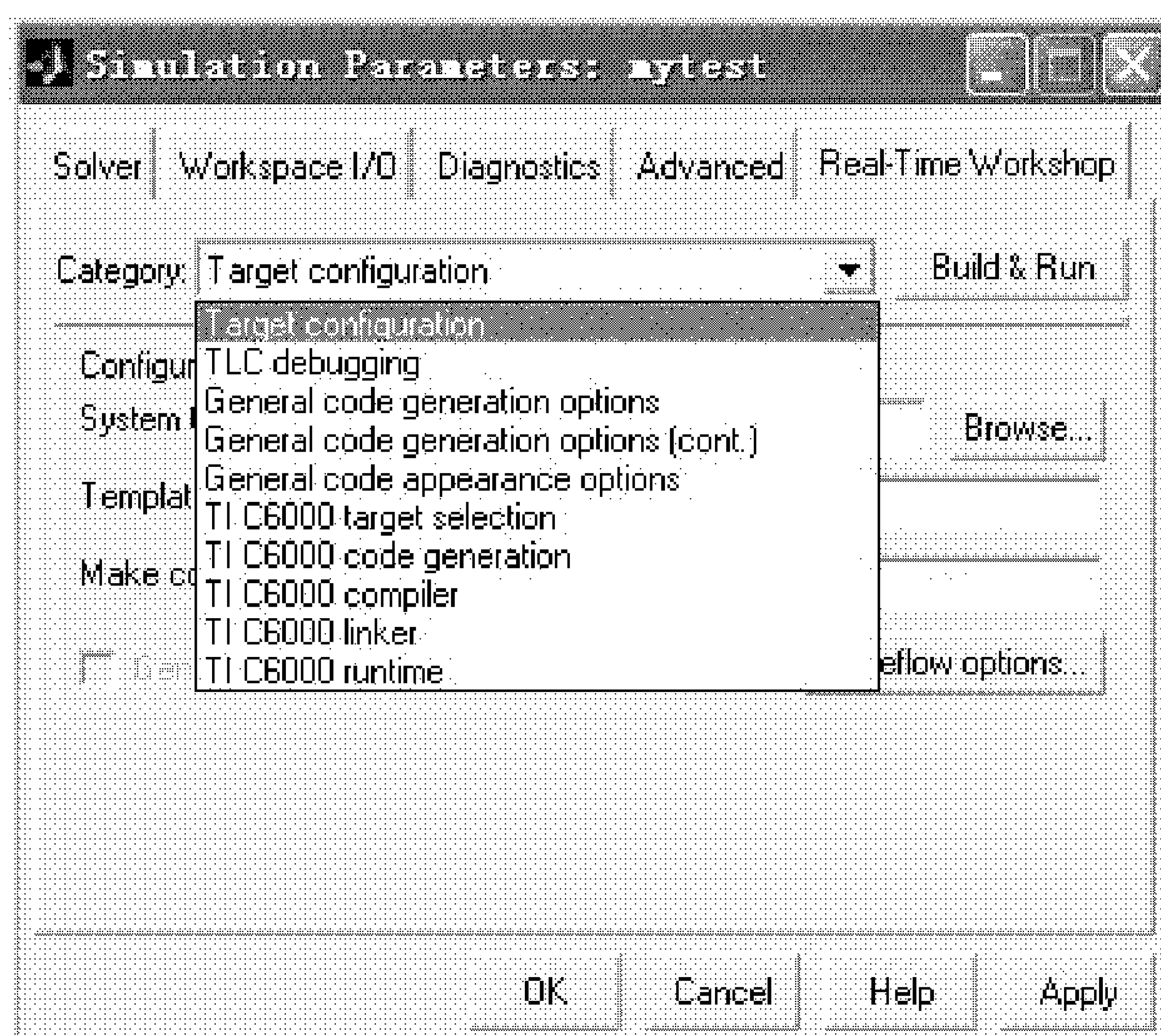


图 6.3　Real－Time Workshop 选项面板

在 Real－Time Workshop 面板上，Category 列表提供了 10 组选项用以控制 Real－Time Workshop 编译链接和模型实时执行过程，其中前 5 组应用于所有 Real－Time Workshop 目标，因此总是在 Category 中出现，而后 5 组选项是专门用于 ETTIC6000 的，当系统目标文件选择为 ti_c6000.tlc 时才会出现。

Category 列表中的 10 组选项如下：

- Target configuration：选择及配置目标。
- TLC debugging：设置目标语言编译器(TLC)的调试和性能统计选项。
- General code generation options：设置 Real－Time Workshop 源代码产生选项。
- General code generation options(cont.)：设置 Real－Time Workshop 源代码产生选项。
- General code appearance options：源代码和标识符格式设置。
- TI C6000 target selection：选择指定的 TI C6000 目标板及其 DSP。
- TI C6000 code generation：定义 TI C6000 实时代码产生方式。
- TI C6000 compiler：设置 TI C6000 编译器选项。
- TI C6000 linker：设置 TI C6000 链接器选项。
- TI C6000 runtime：设置模型在 DSP 上的运行选项。

下面详细介绍这些选项的内容。

6.2.1　Target configuration 选项

此组选项指定为何种目标生成代码，即是 TI C6000 DSP 还是其它目标。图 6.4 为 Target Configuration 选项面板。

- System target file：选择系统目标文件，单击 Browse 按钮会打开一个 System Target File Browser 窗口，如图 6.5 所示，此窗口中列出了多个系统目标文件，选择 ti_C6000.tlc 作为 ETTIC6000 应用的目标文件。当用户选择好 System Target file，Real－Time Workshop 会自动设置 Template makefile 和 Make command 选项。

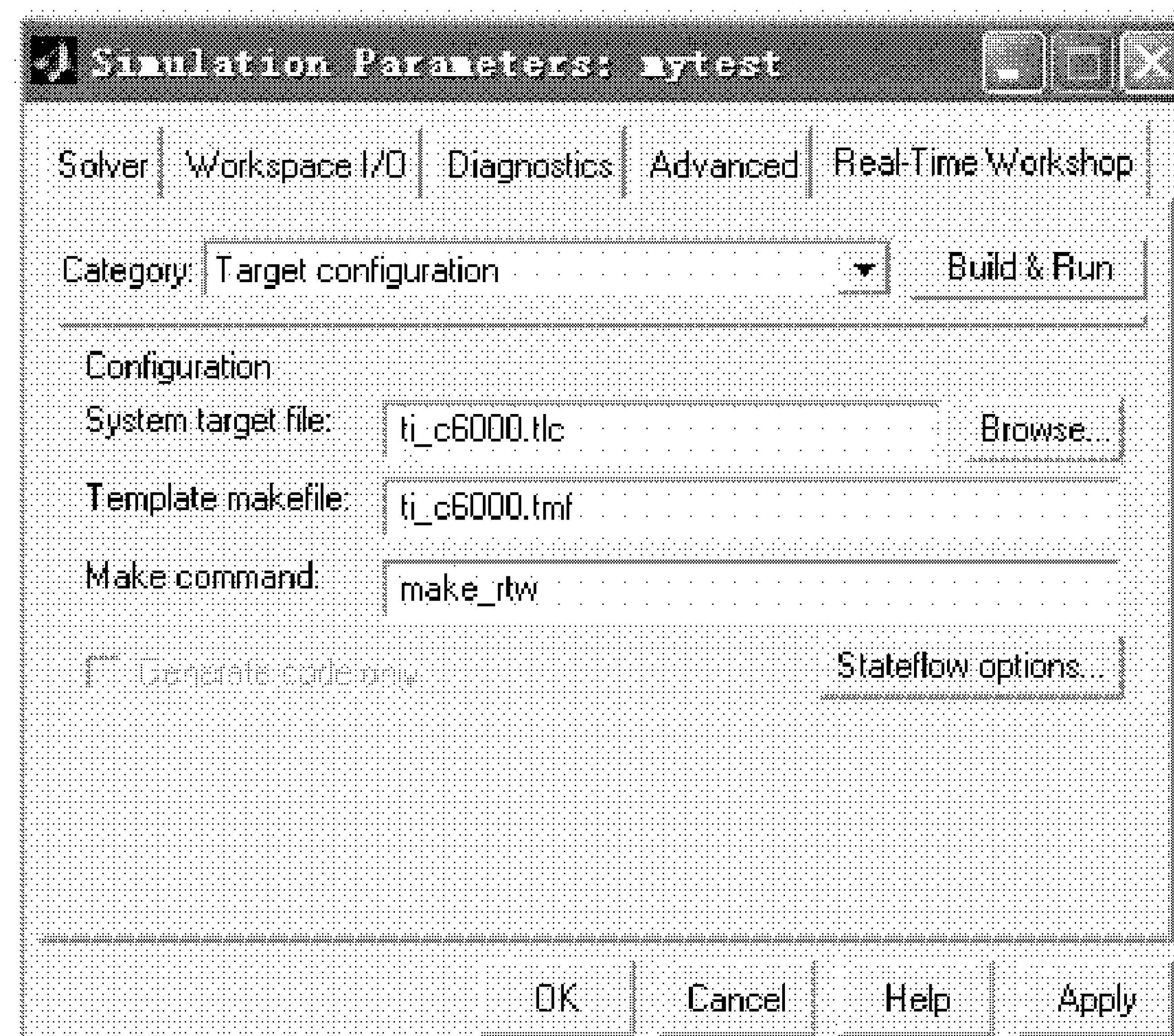


图 6.4　Target Configuration 选项面板

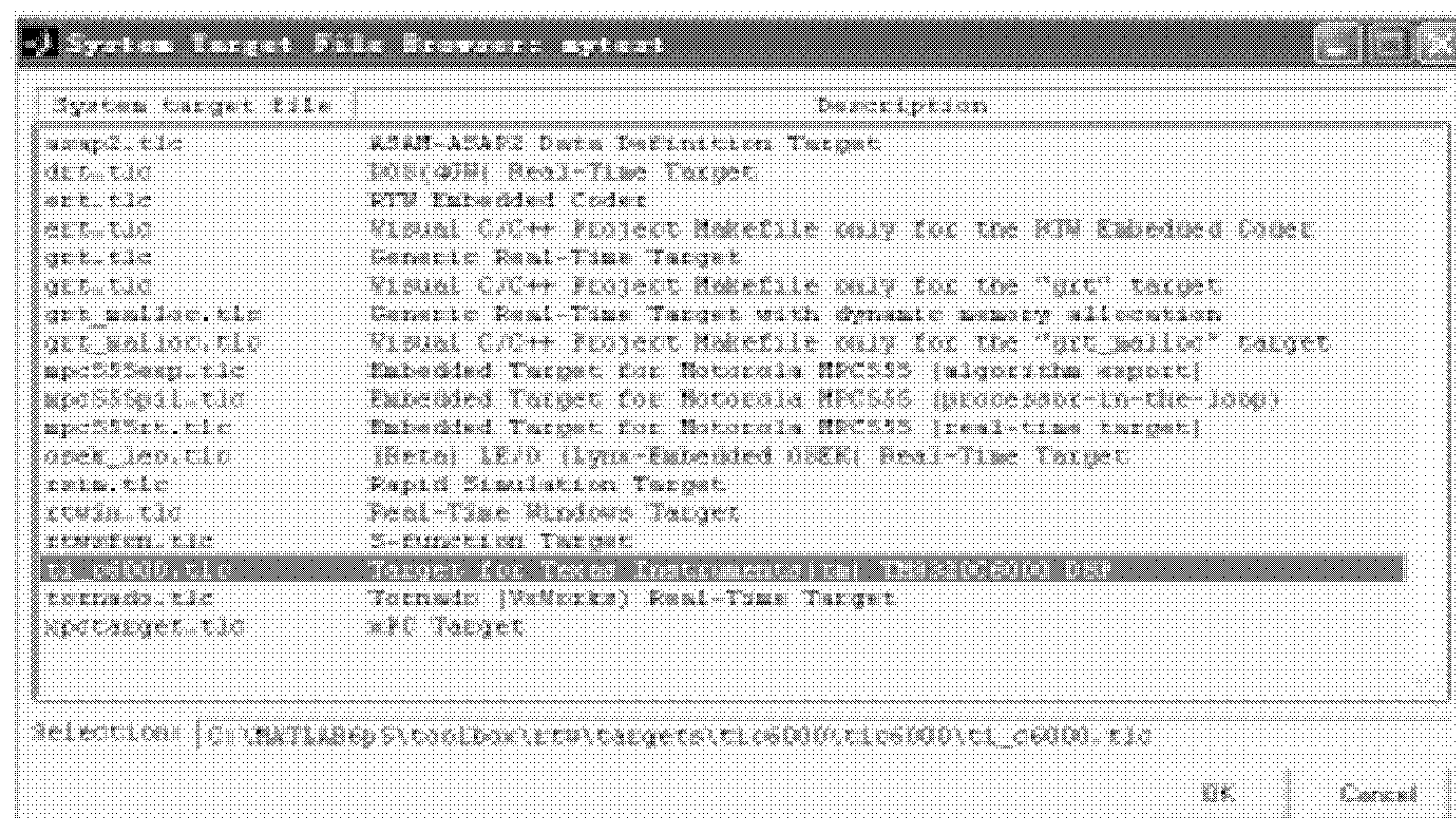


图 6.5　System Target File Browser 窗口

• Template makefile：Real－Time Workshop 利用此 makefile 模板文件产生编译链接所需要的 makefile 文件。

当选择 System target file 为 ti_C6000.tlc 时，Real－Time Workshop 会自动选择 Template makefile 为 ti_C6000.tmf，Make command 为 make_rtw。

在 Real－Time Workshop 编译链接过程中，make_rtw 命令自动释放。make_rtw 从 ti_C6000.tmf 模板 makefile 文件中抽出信息，创建实际的 makefile 文件：*modelname*.mk (*modename* 为模型名，下同)。当 Real－Time Workshop 编译链接 Simulink 模型时，利用实际的 makefile 文件来产生编译链接的目标代码。

• Make command：当选择好 System target file 后，Rea－Time Workshop 会自动选择标准的 Make command 命令 make_rtw，也可以在 Make command 的输入文本框中把 make_rtw 输入进去。

• Generate code only：此选项不应用于 ETTIC6000。

如果要求 ETTIC6000 只产生 C 源代码，而不用调用 CCS 来编译链接及在目标 DSP 上执行代码，应该在 TI C6000 runtime 中选择 Build action 为 Generate code only。

6.2.2　Target language compiler(TLC)debugging 选项

Real－Time Workshop 利用目标语言编译器(TLC)从 *modelname*.rtw 文件生成 C 语言代码。TLC debugger 可帮助用户确定编程错误。利用 TLC debugger 可以完成：观察 TLC 调用堆栈情况；单步执行 TLC 代码，分析或修改变量值。

• Retain .rtw file：通常 build 过程完成之后会删除 *modelname*.rtw 文件，选择此项，*modelname*.rtw 文件就不会被删除。

• Profile TLC：如果选择此项，TLC 会统计分析 TLC 代码的执行性能，并把统计结果生成一个 HTML 格式报告。

• Start TLC debugger when generating code：如果选择此项，在代码产生过程中开始 TLC 调试。

• Start TLC coverage when generating code：如果选择此项，TLC 会产生一个统计表报告，此报告统计在代码产生过程中 TLC 代码的每一行所碰到的次数。

• Enable TLC assertions：如果选择此项，则当用户提供的文件中包含有%assert 命令且其值为 FALSE 时，Real－Time Workshop 会停止代码产生过程。

6.2.3　General code generation 选项

代码产生选项应用于所有目标配置，这些选项分成两组：General code generation options 和 General code generation options(cont.)。

• Show eliminated statements：如果选择此项，程序优化时排除的命令在优化后的代码中作为注解出现。

• Loop rolling threshold：当一个信号或参数的长度超过此门限指定的数值时，会用一个 for 循环来实现对此信号所有元素的操作，否则会对所有元素用分开的程序段进行操作。对所有元素用分开的程序段进行操作，会提高程序的执行效率但代码会变长。

• Verbose builds：如果选择此项，MATLAB 命令窗中会显示代码产生过程信息。

• Generate HTML report：如果选择此项，Real－Time Workshop 会产生一个 HTML 格式的代码产生报告，并在 MATLAB Help 游览器中自动打开它。目标 DSP 不同，报告的内容也不同，但所有报告都包含下列内容：

Summary 段列出了版本和日期信息，TLC 选项和 Simulink 模型设置。

Generated Source Files 段包含从 Simulink 模型中产生的源代码文件表。

• Inline invariant signals：如果选择此项，Real－Time Workshop 会在产生的代码中嵌入不变化的信号(恒定信号)。

• Local block outputs：如果选择此项，模块输出信号将被声明为函数内部信号，否则有可能被声明为全局信号。

• Force generation of parameter comments：如果选择此项，无论 *modelname*_Prm.h 中有多少参数声明，都会加入参数注释(包括参数变量名和所属模块名)，否则只有当参数数目少

于 1000 时才会加入参数注释。

• Buffer reuse：如果选择此项，Real - Time Workshop 会重复使用信号存储器，否则信号存储于惟一的地址空间上。

• Expression folding：是一种代码优化技术，能够大幅度提高生成代码的效率。如果选择此项就会使能这种代码优化技术，对生成的代码进行优化。

• Fold unrolled vectors：如果选择此项，Real - Time Workshop 会对所有信号元素的操作用 for 循环来实现，这样能够减小产生的代码长度。如果关闭此项，对于长度小于 Loop rooling threshold 的信号，Real - Time Workshop 用分开的代码段实现，因此能够提高代码的执行效率。

• Enforce integer downcast：对于 8 bit 操作在 16 bit DSP 上执行或 16 bit 操作在 32 bit DSP 上执行，为了保证仿真结果与实际执行的一致性，8 bit 或 16 bit 整数表达式的结果必须进行转换。选择此项能够确保仿真结果与实际执行的一致性。

6.2.4 General code appearance 选项

此组选项用来控制源代码的格式和标识符的构建。

• Maximum identifier length：用来限制函数名、类型定义名和变量名中字符的长度。默认为 31 个字符。

• Include data type acronym in identifier：如果选择此项，会在生成的程序代码中使用数据类型的缩写词。

• Include system hierarchy number in identifiers：如果选择此项，Real - Time Workshop 会在产生的代码中用表明模块所处嵌套级的字符串作为标识符。

• Prefix model name to global identifiers：如果选择此项，子系统函数名的前面都加一模型名前缀(*modename_*)。

• Generate scalar inlined parameters as：如果参数为标量并且具有常数采样时间，则此项控制参数在代码中的表达，有两种选择：

Literals：参数作为数值常数来表示；

Macros：参数作为变量来表示。

• Generate comments：选择此项会在生成的代码中加入注释，否则代码中的所有注释都被去除。General Code Generation 选项面板中的 Show eliminated statements 和 Force generation of parameter comments 选项用来控制在代码中加入指定类型的注释。

6.2.5 TI C6000 target selection 选项

此组选项用来指定代码产生的目标板和 DSP。

• Code generation target type：选择目标板类型：C6701 EVM 或 C6711 DSK 目标板。

• Board and processor selection method：目标板及其 DSP 的选择方法：Automatic(自动)或 Manual(手动)。

如果选择 Automatic 方法，在每次打开 Simulation Parameters 对话框时，ETTIC6000 会自动运行 ccsboardinfo 函数来决定 CCS 中配置的目标板和 DSP。ETTIC6000 根据 Code

generation target type 中指定的目标板类型识别目标板，并在 Board name 和 Processor number 编辑框中报告板名和 DSP 索引号。如果主机上安装两个或两个以上同类型的目标板，在 Automatic 选择模式下，ETTIC6000 会自动选择 CCS 中分配的第 1 个目标板及其第 1 个 DSP。

如果选择 Manual 方法，需要手工选择 Board name 和 Processor number，用户也需要首先在 MATLAB 命令窗中利用 ccsboardinfo 函数查看 CCS 中定义的目标板及其 DSP 信息，然后再手工指定目标板及其 DSP。

Simulator(软件模拟器)不能在 Automatic 模式下选择，因此如果使用 Simulator，必须进行手工选择。

• Export CCS handle to MATLAB base workspace：利用 Real－Time Workshop 编译链接模型产生 C6000 目标代码时，ETTIC6000 会在 MATLAB 和 CCS 之间建立一个连接对象(调用 MATLAB Link for CCS Development Tools，详见第 5 章)。如果选择此项，ETTIC6000 会向 MATLAB 空间输出此连接对象，此连接对象的句柄名可以由用户在 CCS handle name 文本输入框中指定。

6.2.6 TI C6000 code generation 选项

此组选项定义 TI C6000 目标代码的产生方式。

• Incorporate DSP/BIOS：决定是否在产生的目标代码中集成 DSP/BIOS 功能块。如果选择此项，build 会在产生的代码中插入 DSP/BIOS 功能块和.cdb 文件(包含 DSP/BIOS 配置信息)。

• Profile performance at atomic subsystem boundaries：如果选择此项，并且模型中包含原子级子系统(Atomic Subsystem)，当目标 DSP 运行此模型时，ETTIC6000 会产生一个统计代码执行性能的 HTML 报告。

• Inline DSP blockset functions：决定由 DSP 模块生成的函数体是内嵌在代码中还是通过指针调用。如果选择此项，编译器会在代码中内嵌此函数体。内嵌函数会使代码运行速度提高，但代码会变长。

• Use target specific optimization for speed (allow LSB differences)：决定 ETTIC6000 是否优化产生的代码，使其在目标 DSP 中的运行速度更快。

如果选择此项，ETTIC6000 会对模型产生的代码进行优化，但优化模块的输出结果与仿真结果在最低位(LSB)有所不同，因此用户必须确定优化后的输出结果是否满足需要。如果清除此项，生成代码的输出结果与模型的仿真结果一致。

一种推荐的方法是：先不选择此项，产生代码并且在目标DSP上运行代码，再利用 profile 统计代码的执行性能；当代码运行结果正确后，再选择此项，重新产生代码并在目标 DSP 上运行优化后的代码；最后比较优化前和优化后的 profile 报告，确定优化是否能够提高代码的执行性能，并且查看结果是否满足要求。

6.2.7 TI C6000 compiler 选项

此组选项指定 TI C6000 编译器如何编译代码。如果改变此面板中的设置，这些改变就会变成 CCS 工程中 build 配置选项的一部分，之后可以在 CCS 中编辑并修改它们。

• Optimization level：选择 TI 优化编译器提供的代码优化编译程度。

ETTI C6000 的默认设置为 Function(－o2)。

• Byte order：选择 DSP 的字节组合模式：little－endian 或 big－endian。选择哪一种模式并不影响算法的执行，但会影响不同 DSP 之间的通信。如果目标 DSP 需要与其它 DSP 进行通信，必须选择匹配的字节组合模式。

• Compiler verbosity：用来控制编译器返回的信息量。

Verbose：返回所有的编译器信息；

Quiet：抑制编译器进程信息；

Super Quiet：抑制所有的编译器信息。

• Symbolic debugging：如果选择此项，会产生 C 源程序调试器使用的符号调试指令，并且使能汇编源程序调试。

• Retain .asm files：如果选择此项，Real－Time Workshop 和 ETTIC6000 会在当前路径下保存生成的汇编语言文件(.asm)，否则会删除这些汇编语言文件。

6.2.8 TI C6000 Linker 选项

此组选项用来设置 TI C6000 链接器操作。链接器定义存储器映射并把代码和数据分配到存储器段中。

• Retain .obj files：链接器把多个目标文件(.obj)、库文件等链接成单个可执行的 COFF 文件。

如果选择此项，Real－Time Workshop 和 ETTIC6000 会在当前路径下保存产生的目标文件(.obj)。

• Create.map file：如果选择此项，链接器会产生一个输入/输出存储器段的列表文件 *modelname*.map，并在当前路径下保存此文件。

• Linker command file：指定链接器运行时用到的链接命令文件(.cmd)。链接命令文件包含链接器选项和链接器的输入文件名。有如下选择文件：

User－defined：指定链接器利用用户定义的链接命令文件。用户可以开发自己的链接命令文件，这样会更加满足用户的需要。如果选择此项，用户必须在 User linker command file 文件输入框中输入用户的链接命令文件名(.cmd 后缀)。

Full_memory_map：选择此项，链接器利用大存储器空间模式。在大存储器空间模式下，不会限制非初始化代码段的大小。用户的全局和静态数据能够利用整个存储器空间。

Internal_memory_map：选择此项，链接器利用小存储器空间模式。在小存储器空间模式下，非初始化代码段限制在 32 KB 的存储器空间中。程序中所有全局和静态数据存储空间必须小于 32 KB。小存储器空间模式通过利用近程调用和近程数据可以最优化存储器的使用。

6.2.9 TI C6000 runtime 选项

此组选项用来设置模型在 DSP 上的运行情况。在目标板上运行可执行文件之前，必须先配置好此组选项。

• Build action：指定当用户点击 Build 按钮时，Real - Time Workshop 将如何操作。有四种操作方式可供选择：

Generate_code_only：Real - Time Workshop 只从 Simulink 模型生成 C 语言源代码，而不调用 TI 开发工具来编译链接此 C 源代码，因此也不用安装 CCS 工具。Real - Time Workshop 产生 *modelname*.c、*modelname*.cmd、*modelname*.bld 和其它一些文件，并把这些文件保存在<MATLAB 当前工作路径*modelname*_c6000_rtw\>路径下。Real - Time Workshop 也会产生一个 MS - DOS 批处理文件 *modelname*.bat，此文件中包含 TI C6000 编译器命令行，用来编译产生的 C 代码。此文件中也包含文件和库的路径、默认的编译器设置等。

Creat_CCS_project：Real - Time Workshop 会启动 CCS 并利用模型编译过程生成的文件构建一个新的工程。选择此项，必须指定 CCS 中的目标板。

Build：编译链接生成可执行 COFF 文件，但并不把此可执行文件加载到目标 DSP 中。

Build_and_execute：编译链接生成可执行 COFF 文件，把可执行文件加载到目标 DSP 中并且开始运行。

• Current C6701 EVM CPU clock rate：指定当前 C6701 EVM 板上的时钟频率：25 MHz, 33.25 MHz, 100 MHz 或 133 MHz。尽管 C6701 EVM 板上默认的时钟频率为 100 MHz，但可以利用板上的 DIP 开关或 TI 提供的一种软件工具来修改此时钟频率。C6711 DSK 目标板只有一种 150 MHz 的固定时钟频率，因此不能修改 C6711 DSK 板上的时钟频率。

如果Simulink模型中没有ADC或DAC模块或模型中的数据率发生改变(多速率处理)，Real - Time Workshop会利用定时器来产生中断，驱动模型运行。定时器根据此项设置的CPU时钟频率来计算每一中断需要的CPU时钟数目。因此Current C6701 EVM CPU clock rate 中选择的时钟频率必须与目标板上的实际时钟频率相一致，否则模型运行结果出错。

• Overrun action：设置目标DSP如何响应Overrun(溢出)情况。如果靠定时器产生中断来驱动模型的多速率系统或模型不包含ADC和DAC模块时，Overrun不工作。它有以下选项：

None：忽略遇到的 overrun 情况。

Continue：当 DSP 遇到 overrun 情况时，它打开外部发光二极管并且继续运行。

Halt：当 DSP 遇到 overrun 情况时，它会停止程序的执行同时打开外部发光二极管。

6.3 在生成的目标可执行代码中集成 DSP/BIOS 功能模块

从 Simulink 模型产生 TI C6000 目标代码时，ETTIC6000 能够自动在生成的代码中集成 DSP/BIOS 功能块。关于 DSP/BIOS 的详细介绍及在 CCS 下的应用，请参阅第 3 章，本节主要介绍如何利用 ETTIC6000 在产生的代码中集成 DSP/BIOS 功能块以及在 MATLAB 下如何利用 DSP/BIOS 获取目标程序执行信息。

DSP/BIOS 是一种实时操作系统，能够实时地获取目标 DSP 的信息并对应用程序进行实时分析。DSP/BIOS 包括如下 3 部分：

• DSP/BIOS 配置工具：可以在程序代码中添加和配置任一 DSP/BIOS 功能模块，可以配置中断调度和操作、设置线程优先级和存储器配置等。

• DSP/BIOS 实时分析工具：在 CCS 下可以实时地观察目标 DSP 中程序的执行情况。

• DSP/BIOS 应用程序接口(API)：提供了 150 个 API 函数可供调用，既可以被 C 代码调用也可以被汇编代码调用。

6.3.1　在生成的可执行代码中集成 DSP/BIOS 功能模块

利用 Real－Time Workshop 和 ETTIC6000 从 Simulink 模型生成目标代码时，可以自动在生成的代码中加入 DSP/BIOS 功能块。在生成的代码中自动加入 DSP/BIOS 的方法很简单，可按以下步骤进行：

Simulink 模型窗 → Simulation 菜单 → Simulation parameters 对话框 → Real－Time Workshop 面板 → Category 列表中选 TI C6000 code generation → 选择 Incorporate DSP/BIOS，再设置好 Real－Time Workshop 面板中的其它选项，最后按 Build 按钮，编译链接并生成目标代码。

注意：Build action(位于 TI C6000 runtime 选项组中)中不能选择 Generate_code_only，选择此项后不会在生成的源代码中集成 DSP/BIOS。

ETTIC6000 会在生成的工程中自动加入如下文件：

- *modelname*.cdb：DSP/BIOS 配置文件。
- *modelname*cfg.s62：包含目标程序中用到的 DSP/BIOS 功能块和硬件中断矢量表。
- *modelname*cfg.h62：*modelname*cfg.s62 的头文件。
- *modelname*cfg.h：DSP/BIOS 模块头文件。
- *modelname*cfg_c.c：定义 CSL 结构和属性的 C 源文件。
- *modelname*cfg.cmd：链接命令文件，添加了 DSP/BIOS 库、RTSxxxx.lib 库和运行时支持库。

ETTIC6000 自动完成配置 *modelname*.cdb、DSP/BIOS 功能块和存储器映射的工作，同时删除工程中不再需要的文件(这些文件的功能已经转移到其它文件中)。这些删除的文件包括：

• vectors.asm 文件：定义工程中的硬件中断。使能 DSP/BIOS 后，这些硬件中断矢量表就会自动转移到配置文件的 HWI 段中，因此不再需要 vectors.asm 文件。

• 原先的 *modelname*.cmd 链接命令文件：分配目标 DSP 的存储器段和其它用户链接命令选项。使能 DSP/BIOS 后，所有的存储器段分配都转移到配置文件中。此文件的其它命令选项被集成在一个复合链接命令文件中，复合链接命令文件名仍为 *modelname*.cmd，但此复合链接命令文件的第一行调用 DSP/BIOS 命令文件 *modelname*cfg.cmd，因此利用复合链接命令文件可以在工程中同时应用 DSP/BIOS 命令和用户命令选项。

• 部分*.lib 库文件：提供 DSP 和外围设备需用的库。这些库的内容被集成在 DSP/BIOS 链接命令文件中。

利用 ETTIC6000 在生成的代码中集成 DSP/BIOS 功能时，并不是所有的 DSP/BIOS 功能块都能被集成，可以利用 CCS 在代码中集成更多需要的 DSP/BIOS 功能块。

ETTIC6000 可以在代码中自动集成的 DSP/BIOS 功能块包括：

• LOG 功能块：记录事件和信息。
• STS 功能块：统计代码的执行性能。

- CLK 功能块：管理时钟。
- HWI 和 SWI 功能块：硬件中断和软件中断，控制程序的执行。

6.3.2 统计代码的执行性能

在目标代码中加入 STS 功能块，可以统计代码段的执行性能。

利用 ETTI C6000 统计(Profile)代码段的执行性能，具体实现步骤如下。

1．使能 DSP/BIOS 和代码性能统计

Simulink 模型窗口→Simulation parameter 对话框→Reat－Time Workshop 面板→从 Category 列表中选择 TI C6000 code generation→选择 Incorporate DSP/BIOS 和 Profile performance at atomic subsystem boundaries。

2．在 Simulink 模型中创建原子级子系统

在 ETTIC6000 下，利用 Profile 进行代码性能统计是针对原子级子系统(Atomic Subsystem)进行的，因此应首先在模型中创建原子级子系统。创建的步骤是：

(1) 创建子系统：选择需要进行性能统计的模块，然后从 Edit 菜单中选择 Creat subsystem，模块名改变为 subsystem。

(2) 把子系统转换为原子级子系统：右击每一子系统，然后从弹出的菜单中选择 Subsystem parameter…，会弹出一个 Block parameters:Subsystem 对话框，在此对话框中选择 Treat as atomic unit，然后点击 OK 关闭此对话框。

3．编译链接后生成目标代码，统计代码性能

(1) 设置好 Real－Time Workshop 中的其它选项并编译链接模型，生成 TI C6000 目标代码。

单击 Real－Time Workshop 中的 Build 按钮，生成目标代码并加载到目标 DSP 中运行。运行一段时间后，停止程序执行，并把子系统性能统计结果返回到 MATLAB 中的一个 HTML 报告中。

(2) 利用 MATLAB 函数打开性能统计报告。

cc=ccsdsp：创建 CCS IDE 连接对象。

profile(cc, 'report')：打开性能统计报告。

对 ccsdsp 和 profile 函数的详细介绍，请参考第 5 章。

4．性能统计 HTML 报告内容

报告标题部分包括：模型名、目标板和报告日期。

报告的正文部分包括：

- Overall CPU statistics：从程序开始执行到停止这段时间内统计的整个 CPU 的运行情况。
- Summary of subsystem profiling：列出模型中所有子系统名及其统计的代码性能。
- Profiled simulink subsystems：按子系统名分别列出每一子系统的统计性能，包括此子系统使用的 STS 功能块名。
- STS Objects：列出了代码中所有 STS 功能块名及其统计结果。

6.4 利用 FDATool 工具设计滤波器

信号处理工具箱中的 FDATool 是一种图形化的滤波器设计与分析工具。利用 FDATool 能够快速设计 FIR 或 IIR 滤波器，并可以画出滤波器的幅度/相位响应和零极点分布等。

在 MATLAB 命令窗中输入 fdatool 命令，会打开 FDATool 设计界面，如图 6.6 所示。

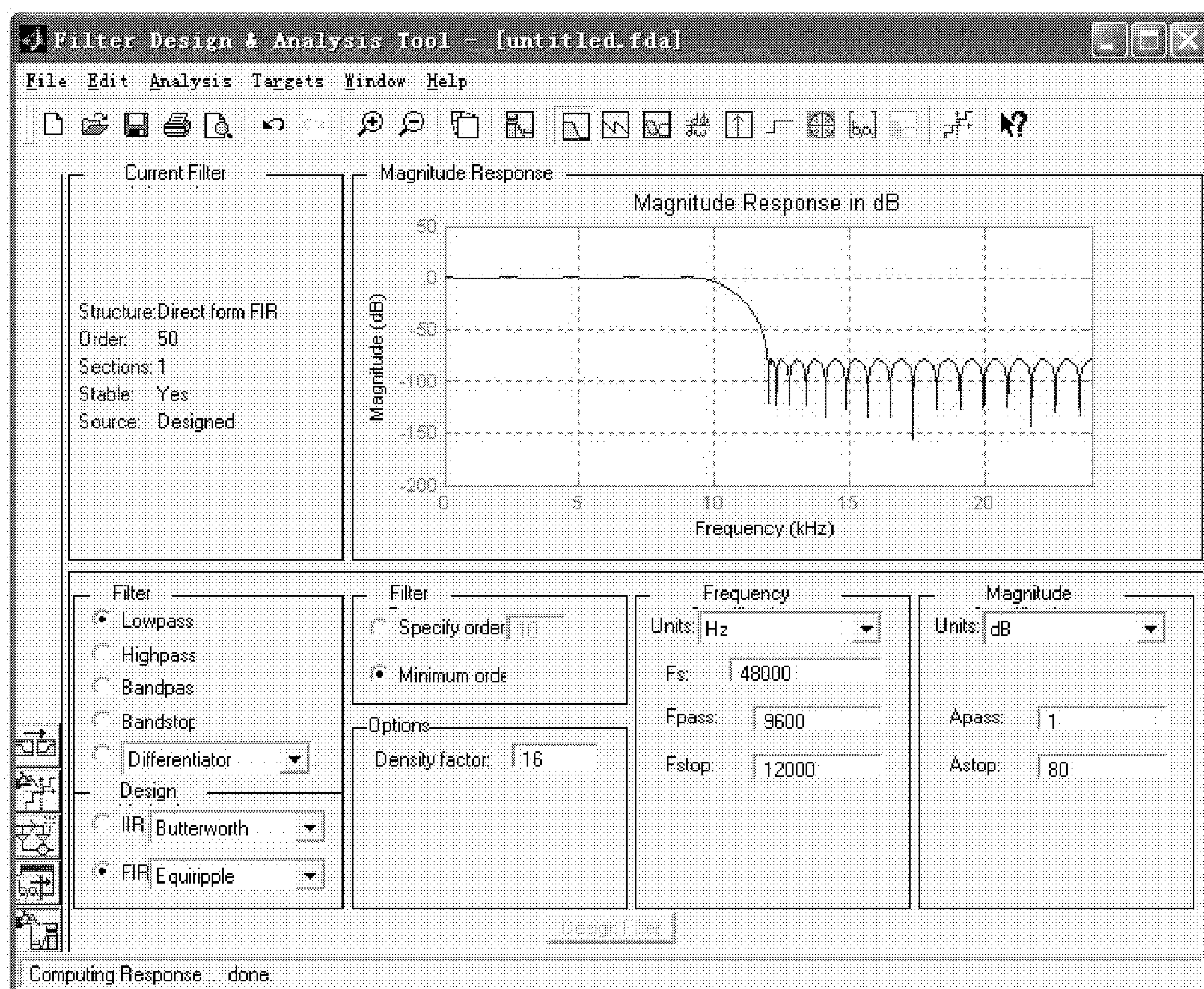


图 6.6 FDATool 设计界面

CCSLink 把 FDATool 与 CCS 连接在一起，在 FDATool 中设计好滤波器后，可以把滤波器系数输出到一个 C 语言头文件或直接输出到 DSP 的存储器中。利用 FDATool 和 CCSLink，可在 FDATool 中调整滤波器系数并在目标 DSP 上进行快速测试。

6.4.1 从 FDATool 向 CCS 输出滤波器系数

有两种方式可以从 FDATool 向 CCS IDE 输出滤波器系数：产生一个 C 语言头文件或直接向目标 DSP 的存储器中写入滤波器系数。

从 FDATool 设计界面的 Targets 菜单中选择 Export to Code Composer Studio(tm)IDE，会弹出一个对话框，如图 6.7 所示。对于不同的滤波器结构，此对话框有所不同。

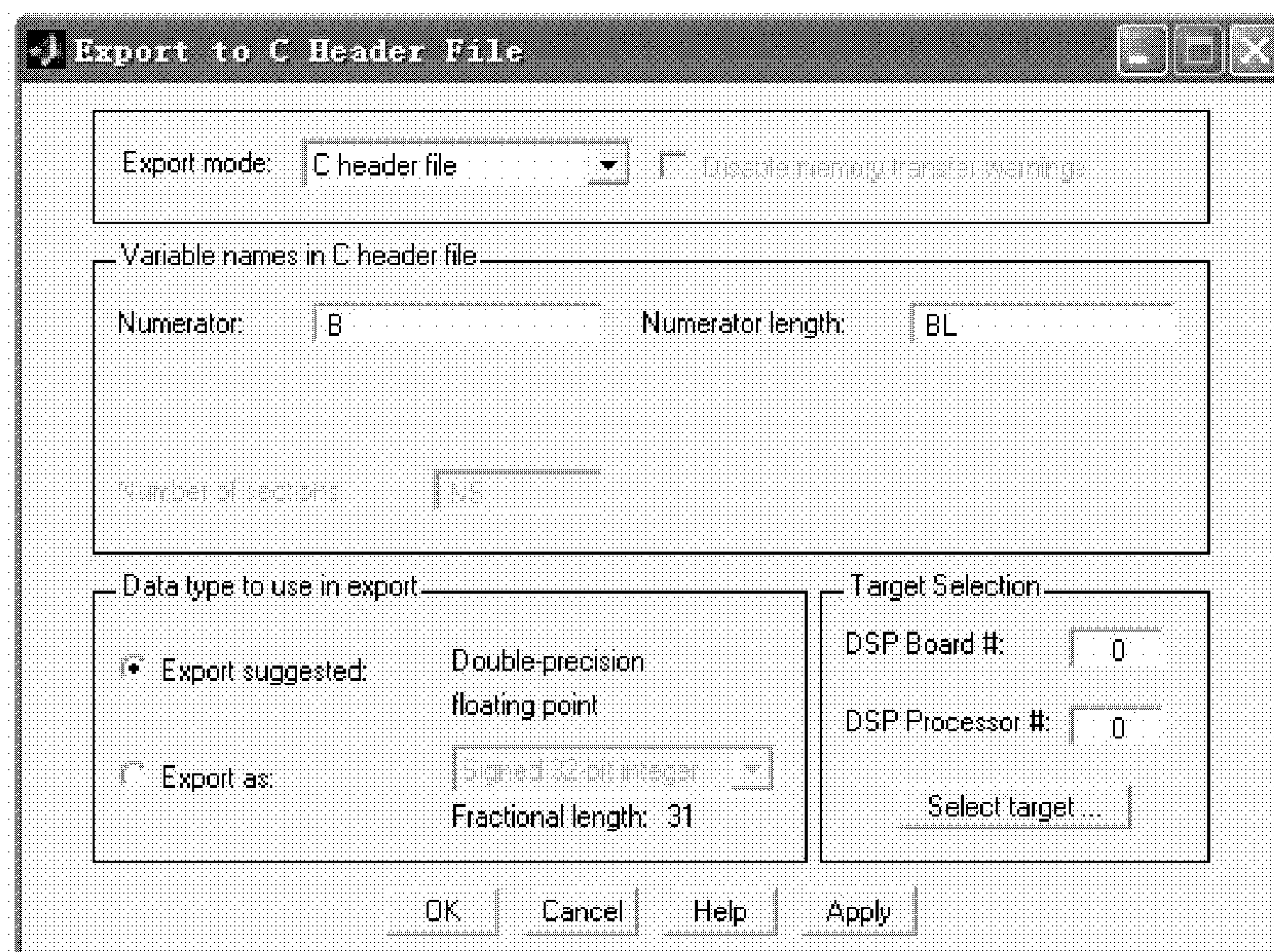


图 6.7　Export to C Header File 对话框

- Export mode：选择输出模式，即 Generate C header file 或 Write directly to memory。

Generate C header file：产生一个 C 语言头文件，并把此头文件添加到目标工程中，然后编译链接工程生成可执行代码。当可执行代码加载到目标 DSP 时会给滤波器系数分配一个静态存储空间。也可以重新编辑此头文件，给滤波器系数分配更大的存储空间。下面的 C 语言程序段为 FDATool 产生的一个滤波器系数的头文件例子。

```
/*
* Filter Design and Analysis Tool-Generated Filter Coefficients-C Source
*   Generated by MATLAB-Signal Processing Toolbox
*/
/* General type conversion for MATLAB generated C-code   */
  #include "tmwtypes.h"
/*
* Expected path to tmwtypes.h
* C:\MATLAB6p5\extern\include\tmwtypes.h
*/
/*
 * Warning-Filter coefficients were truncated to fit specified data type.
 *     The resulting response may not match generated theoretical response.
 *     Use the Filter Design & Analysis Tool to design accurate fixed - point
 *     filter coefficients.
 */
    const int BL = 51;
```

```
const real32_T B[51] = {
  - 0.0009221752407, - 0.002739581279, - 0.002550365636, 0.003556214506, 0.0135493679,
   0.01731581613, 0.007708383258, - 0.006493070628, - 0.007680451497, 0.006085659377,
   0.01387813967, 0.0003968279634, - 0.0168728251, - 0.008919393644, 0.01741195098,
   0.02075459249, - 0.01226045005, - 0.03423506767, - 0.001065741177, 0.04777792096,
   0.02739609033, - 0.05934976414, - 0.08232598752, 0.06715221703, 0.3100234866,
   0.4300875962, 0.3100234866, 0.06715221703, - 0.08232598752, - 0.05934976414,
   0.02739609033, 0.04777792096, - 0.001065741177, - 0.03423506767, - 0.01226045005,
   0.02075459249, 0.01741195098, - 0.008919393644, - 0.0168728251, 0.0003968279634,
   0.01387813967, 0.006085659377, - 0.007680451497, - 0.006493070628, 0.007708383258,
   0.01731581613, 0.0135493679, 0.003556214506, - 0.002550365636, - 0.002739581279,
  - 0.0009221752407
    };
```

Write directly to memory：选择此项前必须在目标 DSP 中已分配好滤波器系数的存储空间。在 FDATool 中调整滤波器系数，并把更新的滤波器系数直接写到已分配好的存储空间中(覆盖先前的滤波器数据)。

把滤波器系数直接写入目标存储器时，必须保证给滤波器系数分配了足够的存储器空间，否则输出的滤波器系数就会占据其它的存储器空间并由此带来不可预料的结果。因此应注意以下几点：不要修改滤波器的输出数据类型；把 FIR 滤波器改变为 IIR 滤波器、提高滤波器的阶数或节数都会增大滤波器的存储空间。

• Variable names in C header file(当选择滤波器系数输出到 C 头文件时)：

当把滤波器系数输出到一个 C 头文件时，此头文件中包含滤波器系数变量。把生成的可执行文件加载到目标 DSP 后，这些滤波器系数变量会出现在程序的符号表中。需要在此对话框中指定这些变量名。

• Variable names in target symbol table(当选择滤波器系数直接输出到存储器中时)：

当把滤波器系数直接输出到目标存储器中时，需要给滤波器系数分配存储空间，这些存储空间对应于滤波器系数变量。因此需要在此对话框中指定滤波器系数变量名。

需要指定的滤波器参数(对于不同的滤波器结构，其参数有所不同)如下：

Numerator：指定滤波器分子系数对应的变量名。

Numerator length：指定滤波器分子系数长度对应的变量名。

Denominator：指定滤波器分母系数对应的变量名。

Denominator length：指定滤波器分母系数长度对应的变量名。

Number of sections：指定滤波器节数对应的变量名(如果滤波器只有 1 节，此项不激活)。

• Data type to use in export：选择滤波器系数输出的数据类型。

Export suggested：推荐的输出数据类型。

Export as：指定输出数据类型，可以支持：Signed integer(8、16、32 位符号整数)，Unsigned integer(8、16、32 位无符号整数)，Double－precision floating point(双精度浮点数)，Single－precision floating point(单精度浮点数)。

- Target selection：选择目标板及其 DSP。

可以直接在 DSP Board#：和 DSP Processor#：的文本输入框中输入目标板及其目标 DSP 的索引号(CCS 配置程序会给每一个安装在主机上的目标板及其 DSP 都分配一个索引号)。也可以利用 Select target…按钮来选择目标板和目标 DSP，单击 Select target…按钮(会自动调用 ccsboardinfo 函数来得到 CCS 中的目标板信息)，会弹出一个 Selection Utility 对话框，在此对话框中选择指定的目标板和目标 DSP。

6.4.2 从 FDATool 向 CCS 输出滤波器系数的操作步骤

1. 利用 C 语言头文件输出滤波器系数

(1) 打开 FDATool。

在 MATLAB 命令窗中输入 fdatool 命令，打开 FDATool 工具的设计界面。

在 FDATool 中设计好指定的滤波器。

(2) 打开 CCS 输出滤波器系数对话框。

从 Target 菜单选择 Export to Code Composer Studio(tm)IDE，打开输出滤波器系数对话框。

(3) 选择输出模式。

在 Export mode 中选择 Generate C header file。

(4) 指定滤波器系数变量名。

在 Numerator, Denominator, Numerator length, Denominator length 和 Number of section 等文本输入框中输入指定的变量名。

(5) 选择输出数据类型。

选择推荐的输出数据类型(Export suggested)或用户指定的输出数据类型(Export as:)。

(6) 选择目标板及目标 DSP。

可以直接在 DSP Board#和 DSP Processor#文本输入框中输入目标板和 DSP 索引号，也可以利用 Select target…选择目标板和 DSP。

(7) 产生一个 C 语言头文件。

单击输出对话框中的 Apply 按钮，产生 C 语言头文件。输入文件名和路径保存此头文件。

(8) 把头文件添加到工程中，编译链接生成可执行文件。

把可执行文件加载到目标 DSP 后，会给头文件中的滤波器系数分配一个静态存储空间。滤波器系数就放入此存储空间中。

2. 直接把滤波器系数输出到 DSP 的存储器中

(1) 在 FDATool 中调整设计的滤波器系数。

(2) 重新选择输出模式为 Write directly to memory。

(3) 指定滤波器系数变量名与 C 头文件中的变量名相同。

(4) 单击输出对话框中的 Apply 按钮，滤波器系数就直接输出到目标 DSP 的存储器中，可以在 FDATool 中继续优化滤波器系数并把更新的系数直接写到 DSP 的存储器中。

6.5　C6000lib 模块库

ETTIC6000 提供了多个 Simulink 模块，分别包含在 C6711 DSK Board Support、C6701 EVM Board Support、C62x DSP Library 和 RTDX Instrumentation 四种模块库中。在 MATLAB 命令窗中输入 c6000lib 命令，就会出现这四种模块库及其所包含的模块(如图 6.2 所示)。下面详细介绍这些模块的功能及使用方法。

6.5.1　C6711 DSK Board Support 模块库

1．C6711 DSK ADC 模块

功能：配置 TI C6711 DSK 板上的编/解码器，把模拟输入信号转换为数字信号输出。

说明：利用此模块从外部信号源(信号产生器、频率产生器或音频设备)获取模拟信号并把模拟信号转换成 DSP 的数字输入信号。

应用此模块之前必须进行参数设置，双击模型中的此模块，弹出 Block Parameters:ADC 对话框，如图 6.8 所示。

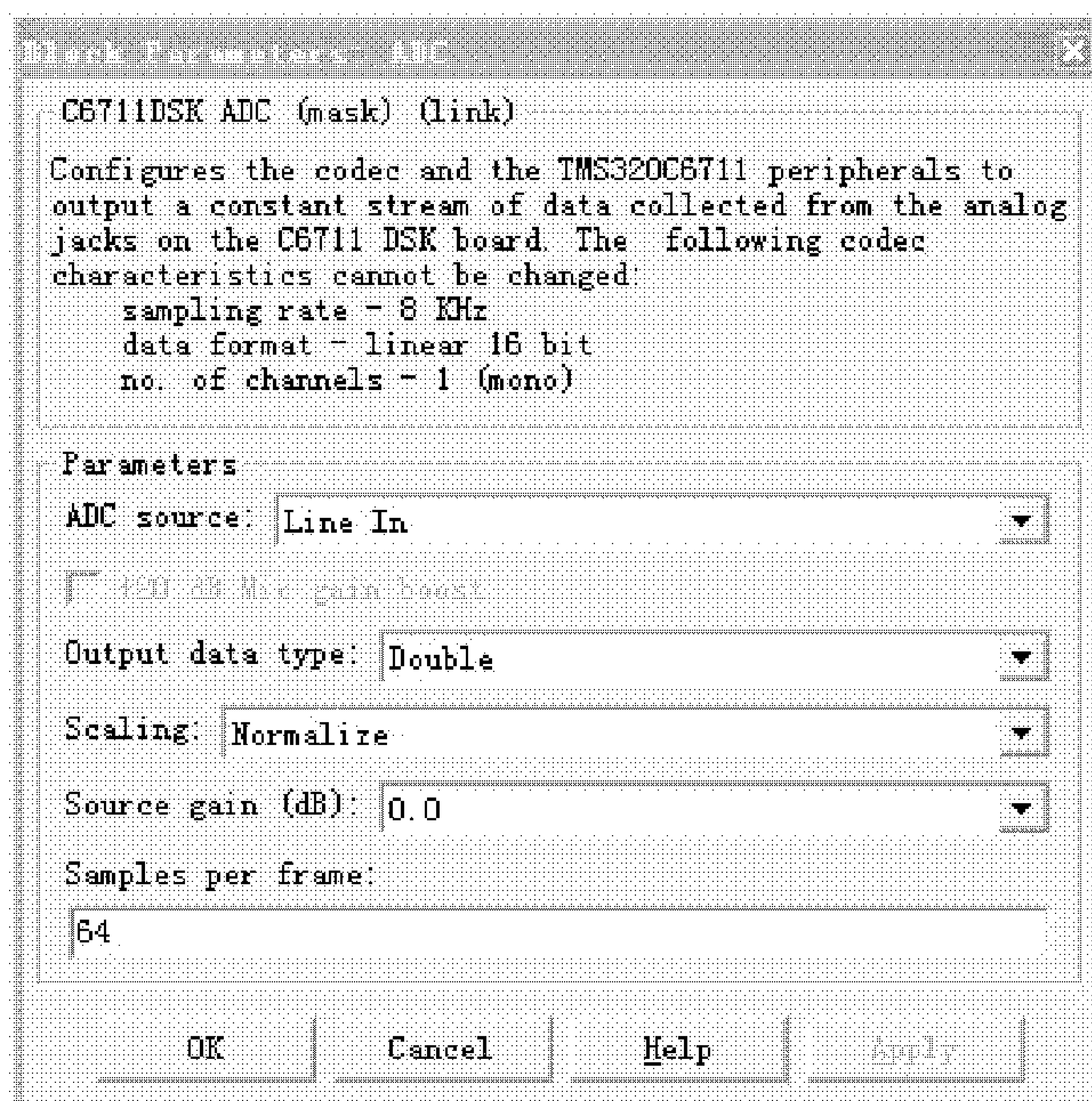


图 6.8　C6711 DSK ADC 模块的参数对话框

- ADC source：编/解码器的输入信号源，有以下三种选项：

Line In：编/解码器从板上的 LINE IN 接线头接收输入信号。

Mic In：编/解码器从板上的麦克风接线头(MIC IN)接收输入信号。

Loopback：把编/解码器输出的模拟信号反馈到编/解码器的输入端，用在反馈应用系统中。

• +20 dB Mic gain boost：当 ADC source 为 Mic In 时，选择此项会把输入编/解码器的信号增益提高 20 dB。

• Output data type：选择编/解码器输出的数据类型，可以支持 Double(双精度浮点数)、Single(单精度浮点数)和 Integer(整数)。

• Scaling：对编/解码器的输出浮点数字信号进行定标，有两种选项：Normalize(归一化到[- 1.0,+1.0]之间)或 Integer Value(实际数值，[- 32 768.0, +32 767.0])。

• Source gain(dB)：指定对编/解码器输入信号的增益大小(单位：分贝)。可以选择[0.0:1.5:12.0]中的任意数值。

• Samples per frame：指定每帧的采样数。把输入信号的采样按帧方式输出，此项指定每帧的采样数。信号的吞吐量保持不变，因此帧率等于采样率(8 K)除以每帧的采样数。

编/解码器输出信号的采样率固定为 8 K，输出编码格式固定为 16 位线性编码，不能修改。

2．C6711 DSK DAC 模块

功能：配置 C6711 DSK 板上的编/解码器，把数字输入信号转变为模拟信号输出。

说明：利用此模块把数字信号转变成模拟信号并输出到 C6711 DSK 板的音频插孔上。

双击此模块，弹出模块参数对话框如图 6.9 所示。

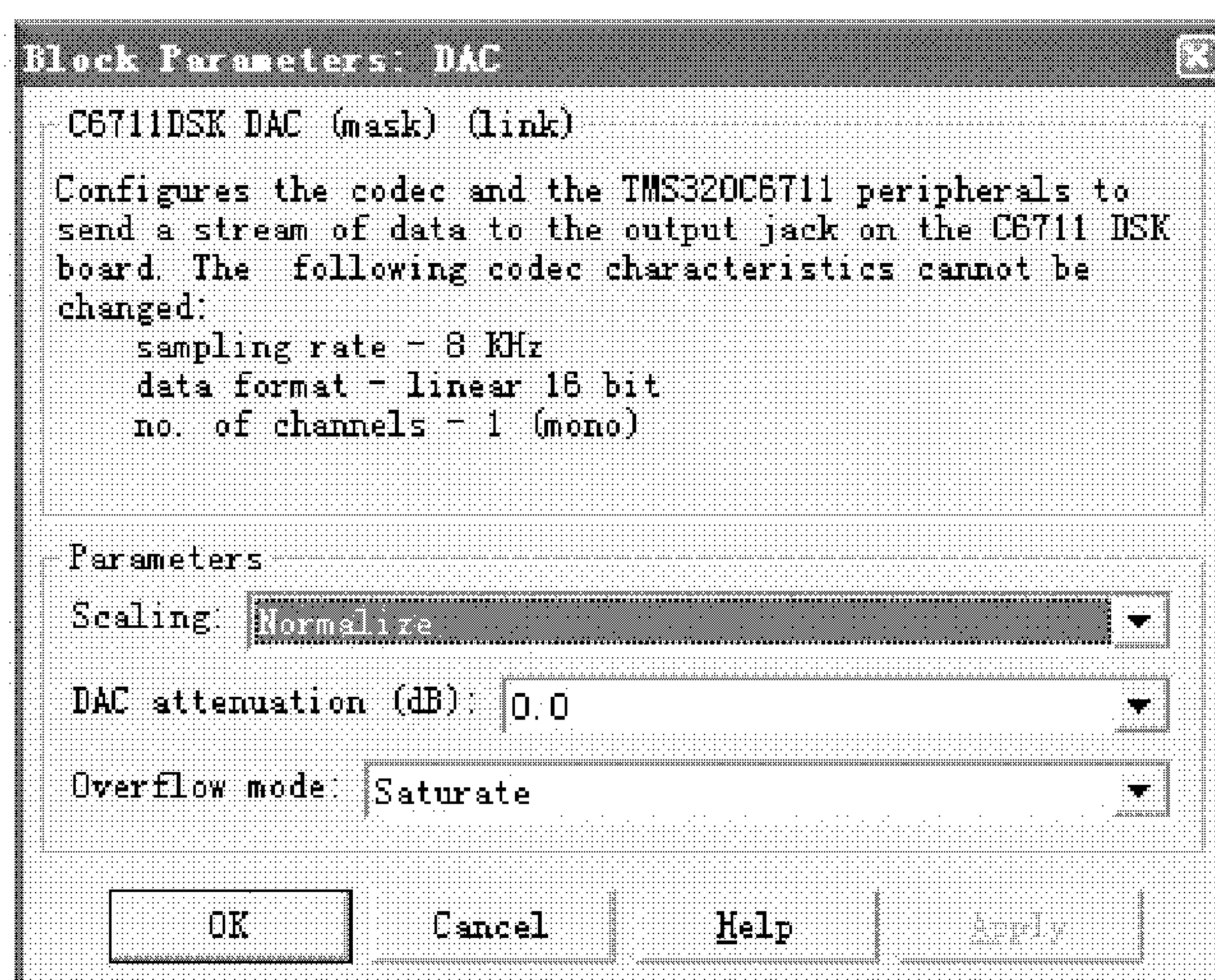


图 6.9　C6711 DSK DAC 模块的参数对话框

• Scaling：对输入编/解码器的浮点数字信号进行定标，有两种选项：Normalize(归一化到[- 1.0,+1.0]之间)或 Integer Value(实际数值，[- 32 768.0, +32 767.0])。在同一 Simulink 模型中，ADC 和 DAC 模块的 Scaling 参数选择应相同。

• DAC attenuation(dB)：指定编/解码器的输出模拟信号的衰减量(单位：分贝)。可以选择[0.0:1.5:36.0]中的任意数值。

• Overflow mode：当编/解码器的输入信号超过 Scaling 参数指定的数值范围时，编/解码器根据此项的设置对输入信号进行处理。有以下两种选项：

Saturate：当输入信号超过指定的数值范围时，对输入信号进行限幅，即对于超过数值范围的信号，用此数值范围内的最大值或最小值来代替。

Wrap：当输入信号超过指定的数值范围时，把超过的信号折叠(即平移)到指定的数值范围内。

3．C6711 DSK DIP Switch 模块

功能：模拟或直接读 C6711 DSK 板上的 DIP 开关状态。

说明：把目标程序的运行方式与 C6711 DSK 板上的 DIP 开关设置对应起来，因此可以通过修改 DIP 开关设置来改变程序的运行情况。此模块还提供了可以模拟板上 DIP 开关状态的参数设置，因此此模块也可应用于 Simulink 模型的仿真阶段。

双击模型中的此模块，打开模块参数对话框，如图 6.10 所示。

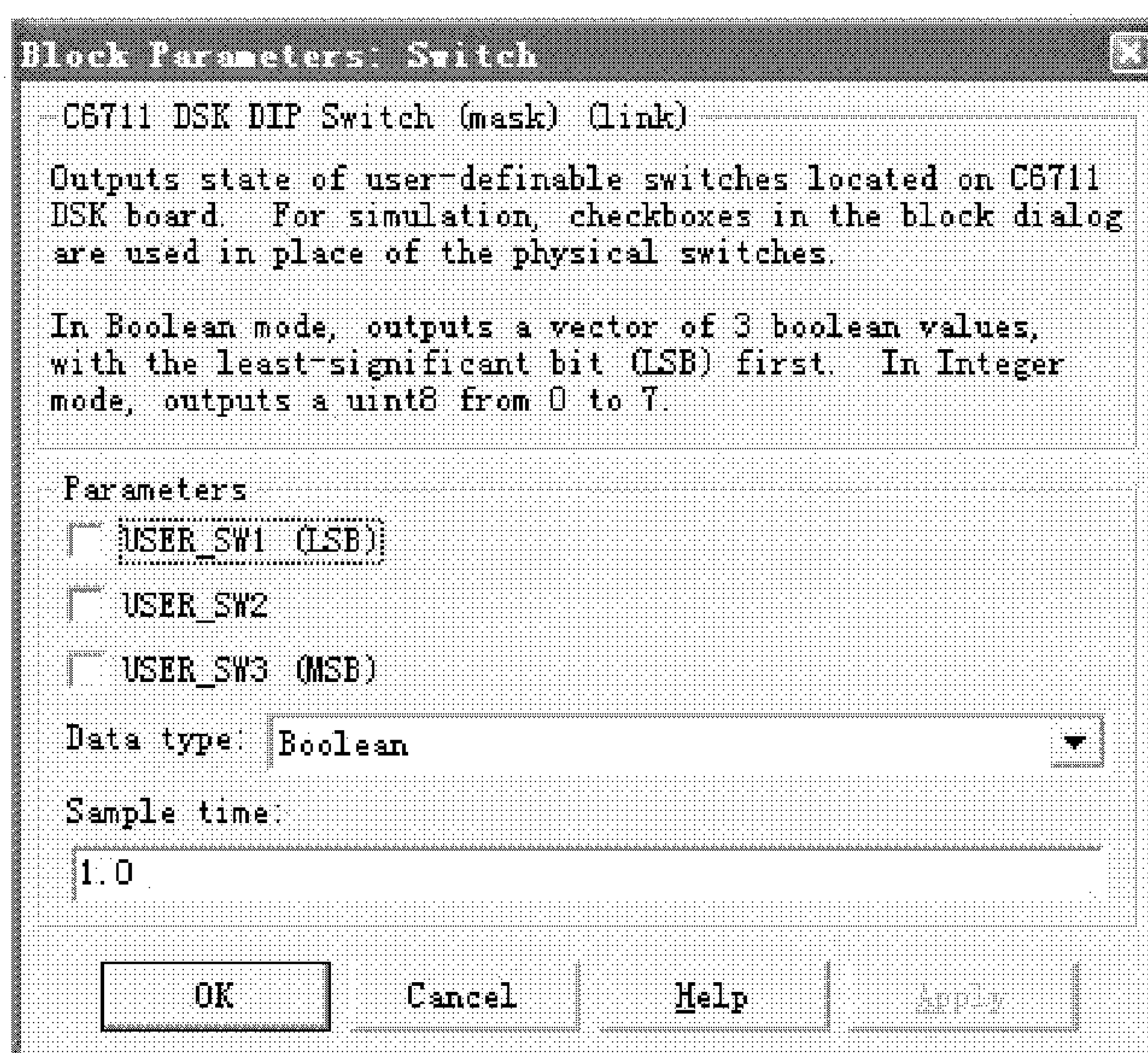


图 6.10　C6711 DSK DIP Switch 模块的参数对话框

- USER_SW1，USER_SW2，USER_SW3：模拟 C6711 DSK 板上对应的 DIP 开关状态。选择表示 on(或 1)，清除表示 off(或 0)。这些参数只用于 Simulink 仿真阶段，而当程序在目标 DSP 上运行后，会忽略掉这些参数而直接读板上的 DIP 开关状态。
- Data type：指定此模块如何解释 DIP 开关状态。有如下两种选项：

Boolean：作为 3 位逻辑串，USER _SW1 为最低位，USER_SW3 为最高位。

Integer：把逻辑串转换为对应的十进制数值。

- Sample time：指定两次读 DIP 状态的时间间隔(单位：秒)。

4. C6711 DSK LED 模块

功能：控制 C6711 DSK 板上的发光二极管(LED)。

说明：此模块可以用来指示目标程序的运行过程或状态。向此模块发送一个非零标量数值，C6711 DSK 板上的三个 LED 会同时打开；向此模块发送一个标量数值 0，C6711 DSK 板上的三个 LED 会同时关闭。LED 会一直保持它的状态(关闭或打开)，直到接收到一个改变其状态的标量数值为止。复位 C6711 DSK 板，这三个 LED 都会关闭。

5．C6711 DSK RESET 模块

功能：把 C6711 DSK 板复位到它的初始状态。

说明：此模块在模型中不与其它任何模块连接。双击此模块，将调用CCS中的软件复位函数来复位C6711 DSK目标板。

6.5.2　C6701 EVM Board Support模块库

1．C6701 EVM ADC模块

功能：配置TI C6701 EVM板上的编/解码器，把输入模拟信号转换成数字信号输出。

说明：利用此模块从外部信号源(信号产生器、频率产生器或音频设备)获取模拟信号并把模拟信号转换成DSP的数字输入信号。

双击Simulink模型中的此模块，弹出模块参数对话框，类似于图6.8。

• ADC source：编/解码器的输入信号源，有以下三种选项：

Line In：编/解码器从板上的LINE IN接线头接收输入信号。

Mic In：编/解码器从板上的麦克风接线头(MIC IN)接收输入信号。

Loopback：把编/解码器输出的模拟信号反馈到编/解码器的输入端，用在反馈系统中。

• +20 dB Mic gain boost：当ADC source为Mic In时，选择此项会把编/解码器的输入信号增益提高20 dB。

• Stereo：指定编/解码器的音频输入为单声道(单通道模拟输入，左通道)或立体声(双通道模拟输入，左右通道)。编/解码器利用32位字来存储数字化的输入信号。单通道和双通道数据在32位字中的位置，如表6.1所示。

表6.1　单通道和双通道数据在32位字中的放置

Format	Left and Mono Channel (32位字的前16位)	Right and Stereo Channel (32位字的后16位)
16 - bit mono	0xLLLL	0x0000
16 - bit stereo	0xLLLL	0xRRRR
8 - bit mono	0xLL00	0x0000
8 - bit stereo	0xLL00	0xRR00
4 - bit mono	0xL000	0x0000
4 - bit stereo	0xL000	0xR000

• Sample rate(Hz)：指定编/解码器对模拟输入信号的采样频率(单位：Hz)。采样频率的范围从5500 Hz到48 000 Hz，但必须从列表中选择某一频率。

• Codec data format：配置编/解码器输出数字信号的编码格式。支持以下五种编码格式：16 - bit linear(16位线性编码)、8 - bit linear(8位线性编码)、8 - bit A - law(8位A律编码)、8 - bit mu - law(8位μ律编码)和4 - bit IMA ADPCM(4位修正的差分PCM编码)。

• Output data type：选择编/解码器的输出数据类型，可以支持Double(双精度浮点)、Single(单精度浮点)和Integer(整数)。

• Scaling：对编/解码器的输出浮点数字信号进行定标。有两种选项：Normalize(归一化到[－1.0,+1.0]之间)或Integer Value(实际数值)。

Codec data format, Output data type和Scaling联合控制编/解码器的输出数字信号的格式

和数值范围，表 6.2 列出了它们对应的数值范围。

表 6.2　由 Codec data fomat、Output data type 和 Scaling 指定的数值范围

Output data type		Integer	Single 或 Double Precision	Single 或 Double Precision
Scaling		—	Normalized	Integer Value
Codec Data Format	8 - bit Unsigned	0～255	- 1.0 to 1.0	0.0～255.0
	16 - bit Linear	- 32 768～32 767	- 1.0 to 1.0	- 32 768.0～32 767.0
	A - law	Unsigned 8 - bit(0～255)	- 1.0 to 1.0	0.0～255.0
	μ - law	Unsigned 8 - bit(0～255)	- 1.0 to 1.0	0.0～255.0
	ADPCM	Unsigned 8 - bit(0～255)	N/A	0.0～255.0

• Source gain(dB)：指定编/解码器输入信号的增益大小(单位：分贝)。可从列表中选择任意增益值。

• Samples per frame：可以把数字信号按帧方式发送出去，此项指定每帧的采样数。信号的吞吐量保持不变，因此帧率等于采样率除以每帧的采样数。

2．C6701 EVM DAC 模块

功能：配置 C6701 EVM 板上的编/解码器，把数字输入信号转换成模拟信号输出。

说明：利用此模块把数字信号转换成模拟信号并输出到 C6701 EVM 板的音频插孔上。

双击模型中的此模块，弹出模块参数对话框，类似于图 6.9。

• Codec data format：指定输入编/解码器的数字信号编码格式。支持以下五种编码格式：16 - bit linear(16 位线性编码)、8 - bit linear(8 位线性编码)、8 - bit A - law(8 位 A 律编码)、8 - bit mu - law(8 位μ 律编码)和 4 - bit IMA ADPCM(4 位修正的差分 PCM 编码)。

在同一模型中，C6701 EVM DAC 模块和 C6701 EVM ADC 模块的 Codec data format 编码格式必须相同。

• Scaling：对输入编/解码器的浮点数字信号进行定标，有两种选项：Normalize(归一化到[- 1.0, +1.0]之间)或 Integer Value(实际数值)。

在同一模型中，C6701 EVM DAC 模块和 C6701 EVM ADC 模块的 Scaling 参数选择应相同。

• DAC attenuation(dB)：指定编/解码器的输出模拟信号的衰减量(单位：分贝)。可以选择列表中的任意数值。

• Overflow mode：当编/解码器的输入信号超过 Scaling 和 Codec data format 指定的数值范围时，编/解码器根据此项的指定对输入信号进行处理。有以下两种选项：

Saturate：当输入信号超过指定的数值范围时，对输入信号进行限幅。

Wrap：当输入信号超过指定的数值范围时，把超过的信号折叠到指定的数值范围内。

3．C6701 EVM DIP Switch 模块

功能：模拟或直接读 C6701 EVM 板上的 DIP 开关状态。

说明：把目标程序的运行方式与 C6701 EVM 板上的 DIP 开关设置对应起来，因此可以通过修改 DIP 开关设置来改变程序的运行方式。此模块还提供了可以模拟板上 DIP 开关状

态的参数设置，因此此模块也可以用于 Simulink 模型的仿真阶段。

双击此模块，打开模块参数对话框。

• USER0, USER1, USER2：模拟 C6701 EVM 板上对应的 DIP 开关状态。选择此项表示 on(或 1)，清除此项表示 off(或 0)。这些参数只用于 Simulink 仿真阶段，而当程序在目标 DSP 上运行后，会忽略掉这些参数而直接读板上的 DIP 开关状态。

• Data type：指定此模块如何解释 DIP 开关状态。有如下两种选项：

Boolean：作为 3 位逻辑串，USER0 为最低位，USER2 为最高位；

Integer：把逻辑串转换成对应的十进制数值。

• Sample time：指定两次读 DIP 状态的时间间隔(单位：秒)。

4．C6701 EVM LED 模块

功能：控制 C6701 EVM 板上的发光二极管(LED)。

说明：此模块可以用来指示目标程序的运行过程或状态。C6701 EVM 板上有两个 LED，一个安装在板的固定架上，称为外部 LED(external，LED0)，另一个安装在板上称为内部 LED(internal，LED1)。应用此模块之前必须先在参数对话框中指定是外部 LED 还是内部 LED。

向此模块发送一个非零标量数值会打开指定的 LED；向此模块发送一个标量数值 0 会关闭指定的 LED。LED 会一直保持它的状态(关闭或打开)，直到接收到一个改变其状态的标量数值为止。

5．C6701 EVM RESET 模块

功能：把 C6701 EVM 板复位到它的初始状态。

说明：此模块在模型中不与其它任何模块连接。双击此模块会调用 CCS 中的软件复位函数来复位 C6701 EVM 目标板。

6.5.3 RTDX Instrumentation 模块库

1．From Rtdx 模块

功能：在目标代码中添加一个 RTDX 输入通道。

说明：在 Simulink 模型中加入此模块，会在编译链接生成的目标代码中插入创建 RTDX 输入通道的指令。利用此通道可以从主机向目标 DSP 发送数据。在 Simulink 仿真阶段，此模块不执行任何操作。

应用 RTDX 模块的操作步骤如下：

(1) 在 Simulink 模型中添加一个或多个 To Rtdx 或 From Rtdx 模块。

(2) 编译链接此模型，生成可执行目标代码。

(3) 加载并运行可执行代码。

(4) 从 MATLAB 环境下使能 RTDX 通道。

(5) 利用 readmsg 和 writemsg 函数(在 MATLAB Link for CCS Development Tools 中)向目标 DSP 发送或从目标 DSP 抽出数据。

双击此模块打开模块参数对话框，如图 6.11 所示。

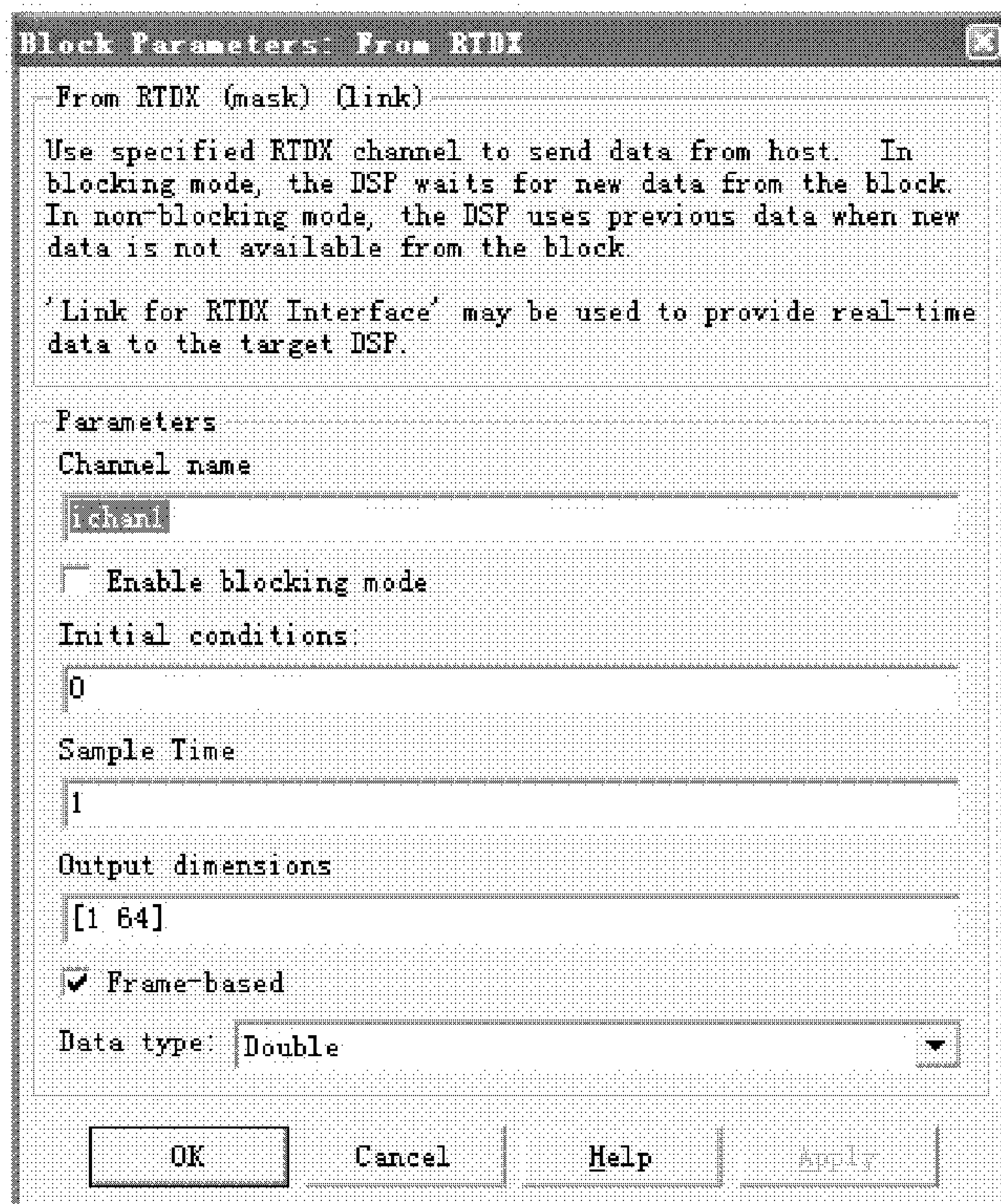


图 6.11　From Rtdx 模块的参数对话框

• Channel name：定义 RTDX 输入通道名。主机利用此通道名来打开、关闭、使能或禁止此通道，向此通道发送数据。

• Blocking：如果选择此项，目标 DSP 会等待新数据的到来，然后再去处理。如果 DSP 需要的新数据没有到来，处理过程会一直等待。

如果清除此项，当新的数据没有到来时，DSP 会利用旧数据继续进行处理。

• Initial conditions：指定 DSP 第一次从 RTDX 输入通道中读到的数据。此项可以输入一个标量值或[ ]。

标量值：利用此标量值初始化输出矩阵(Output dimension，指矩阵的维数)的所有元素。

[ ]：初始化输出矩阵的所有元素都为 0。

如果前面选择了 Blocking，此项不激活。

• Sample time：指定采样 RTDX 的输入数据的时间间隔(单位：秒)。

• Output dimension：指定此模块输出矩阵的维数。例如，输入[1 64]表示输出矩阵的维数为 1×64。

• Frame－based：如果选择此项，模块将采用帧基处理。在帧基处理模式下，帧中的数据会被同时处理，从而可以大大提高程序的处理速度。数据的吞吐量不变，因此帧率等于采样率除以每帧的样本数。

• Data type：设置此模块输出的数据类型，可以支持 Double(双精度浮点，归一化到[-1.0,+1.0]之间)、Single(单精度浮点，归一化到[-1.0, +1.0]之间)、Uint8(8 位无符号整数)、Int16(16 位符号整数)和 Int32(32 位符号整数)。

2．To Rtdx 模块

功能：在目标代码中添加一个 RTDX 输出通道。

说明：在 Simulink 模型中加入此模块，会在编译链接生成的目标代码中插入创建 RTDX 输出通道的命令。利用此通道可以从目标 DSP 向主机发送数据。在 Simulink 仿真阶段，此模块不执行任何操作。

双击此模块，打开模块参数对话框，此模块参数对话框中只有一个参数。

• Channel Name：定义 RTDX 的输出通道名。主机利用此通道名来打开、关闭、使能或禁止此通道，接收此通道中的数据。

6.5.4　TI C62 DSPLIB 模块库

C62x DSPlib 库中的所有模块都对应 C62x 优化的汇编语言函数，这些模块在编译链接时会自动用其对应的汇编语言函数来替代。因此，模型中加入这些模块可以大大提高目标代码的执行速度。

这些定点模块的输入和输出都是定点类型。当与其它模块(Simulink、DSP Blockset 和 Fixed－point Blockset 等中的模块)连接时，必须进行数据类型转换。

当 C62x DSPlib 库中的模块与 Fixed－Point Blockset 中的模块连接时需要设置 Fixed－Point Blockset 模块的数据类型参数和定标(scaling)参数。

部分 DSP Blockset 和 Simulink 模块也可用于定点数据类型，这些模块与 C62x DSPlib 库中的模块进行连接时也必须进行正确的设置。对于不能操作定点数据的 DSP Blockset 或 Simulink 模块，与 C62x DSPlib 库中的模块进行连接时必须经过数据类型转换模块。

• 把定点和非定点模块连接时，中间利用 Fixed－Point Blockset→Data Type 模块库中的 Gateway In 和 Gateway Out 模块进行连接。

• 利用 C62x DSPlib 库中的 Convert Floating－Point to Q.15 和 Convert Q.15 to Floating－Point 模块把浮点类型转换为定点类型或把定点类型转换为浮点类型。

• 利用 Signals and Systems 库的 Data Type Conversion 模块来连接不同的非定点数据类型模块。

• 利用 Fixed－Point Blockset→Data Type 模块库中的 Conversion 和 Conversion Inherited 模块来连接不同的定点数据类型模块。

6.6　由 Simulink 模型生成实时代码过程

本节对应用 ETTIC6000 创建嵌入式实时应用过程进行总结。

1．创建实时 Simulink 模型

创建实时 TI C6000 DSP 处理模型与创建其它 Simulink 模型的方法一样。可利用以下模块创建模型：

• 利用 DSP Blockset 中的模块；
• 利用 Fixed－Point Blockset 中的模块；
• 利用 TI C62x DSPlib 库中提供的定点模块；

- 利用其它的 Simulink 离散时间模块；
- 利用 TI C6701 EVM 和 TI C6711 DSK 板模块；
- 利用其它任何满足要求的模块；
- 利用用户自己创建的模块。

有些模块需要与 MATLAB 工作空间进行通信，在实时模型中加入这些模块会降低实时应用的速度，因为这些模块会占用时间以向 MATLAB 空间发送或接收数据。向 MATLAB 空间发送或从 MATLAB 空间接收数据，可以利用 ETTIC6000 提供的 To Rtdx 和 From Rtdx 模块来实现。表 6.3 列出了在实时模型中建议不要使用的模块。

表 6.3　在实时模型中避免使用的模块

模块名/类	所　属　库
Scope	Simulink,DSP Blockset
To Workspace	Simulink
From Workspace	Simulink
Spectrum Scope	DSP Blockset
To File	Simulink
From File	Simulink
Triggered to Workspace	DSP Blockset
Signal To Workspace	DSP Blockset
Signal From Workspace	DSP Blockset
Triggered Signal From Workspace	DSP Blockset
To Wave device	DSP Blockset
From Wave device	DSP Blockset
To Wave file	DSP Blockset
From Wave file	DSP Blockset

使用模块时还需要设置模块参数。

2．设置仿真参数和 Real - Time Workshop 选项

详见 6.2 节 Real - Time Workshop 的选项设置。

3．编译实时模型

根据 Real - Time Workshop 中的设置有如下 4 种操作模式：

- 只生成源代码。
- 只创建一个新工程。
- 编译链接，生成可执行代码。
- 编译链接，生成可执行代码并加载到目标板，然后开始运行。

6.7　TI C6701 EVM 目标板的应用

TI 公司推出的 C6701 EVM 板及其软件工具可以帮助开发者快速地进行代码开发、调试及验证。ETTIC6000 为 C6701 EVM 板上的 I/O 设备提供了 Simulink 模块库，利用 TI 的 C6701

EVM 开发工具和 ETTIC6000 能够进行快速的原型开发和硬件在线仿真。

本节为使用 TI C6701 EVM 目标板的用户介绍如何在 ETTIC6000 下配置、测试和开发自己的嵌入式实时应用。

6.7.1 TI C6701 EVM 板的配置、验证和测试

1. 配置 TI C6701 EVM 板

安装好 TI C6701 EVM 板及其软件后，还必须设置板上的 DIP 开关，C6701 EVM 板才能在 ETTIC6000 环境下工作。DIP 开关的设置如表 6.4 所示。

表 6.4 在 ETTI C6000 下工作时 C6701 EVM 板上的 DIP 开关设置

开 关	名 称	设 置	作 用
SW2 - 1	BOOTMODE4	On	加载模式设置
SW2 - 2	BOOTMODE3	On	加载模式设置
SW2 - 3	BOOTMODE2	Off	当 SW2 - 5 为 off 时，设置 memory map=1
SW2 - 4	BOOTMODE1	On	加载模式设置
SW2 - 5	BOOTMODE0	Off	当 SW2 - 3 为 off 时，设置 memory map=1
SW2 - 6	CLKMODE	On	设置时钟频率为*4 模式
SW2 - 7	CLKSEL	On	选择晶振 A
SW2 - 8	ENDIAN	On	选择 little - endian 模式
SW2 - 9	JTAGSEL	Off	选择 JTAG
SW2 - 10	USER2	On	用户定义
SW2 - 11	USER1	On	用户定义
SW2 - 12	USER0	On	用户定义

2. 验证 TI C6701 EVM 板是否安装和配置成功

TI 公司提供了一个测试程序用来验证 EVM 板及其软件的工作情况，按下列步骤完成验证：

(1) 启动 CCS。

(2) 选择 Start(桌面开始菜单)→Programs→Code Composer Studio→EVM Confidence Test。

测试程序运行后，测试结果会显示出来。

(3) 观察测试结果，检查是否一切工作正常。

查看测试结果的 DIP 开关设置是否与表 6.4 的 DIP 开关设置一致。如果测试结果的 DIP 开关设置与表 6.4 的 DIP 设置不一致，则需要重新设置 EVM 板上的 DIP 开关，并重新利用此测试程序进行验证，直到结果一致为止。

3. 测试 C6701 EVM 板在 ETTIC6000 环境下是否工作正常

ETTIC6000 提供了几个 Simulink 演示模型，可以用来测试 C6701 EVM 和 C6711 DSK 在 ETTIC6000 环境下的工作情况。在 MATLAB 命令窗中输入：help tic6000，可以显示出这些演示模型名及其说明。其中 c6701evmafxr.mdl 模型用来模拟声音的回音现象，下面利用

此演示模型来测试C6701 EVM板在ETTIC6000环境下是否工作正常。

利用c6701evmafxr.mdl模型进行测试之前，需要把一个麦克风连接到板上的MIC IN接头上，把一个扬声器连接到板上的LINE OUT接头上。当c6701evmafxr.mdl模型在此目标板上运行后，向麦克风说话，可以从扬声器里听到延迟后的话音。按下列步骤完成测试：

(1) 在MATLAB命令窗中输入c6701evmafxr命令，打开c6701evmafxr.mdl Simulink模型窗口，如图6.12所示。

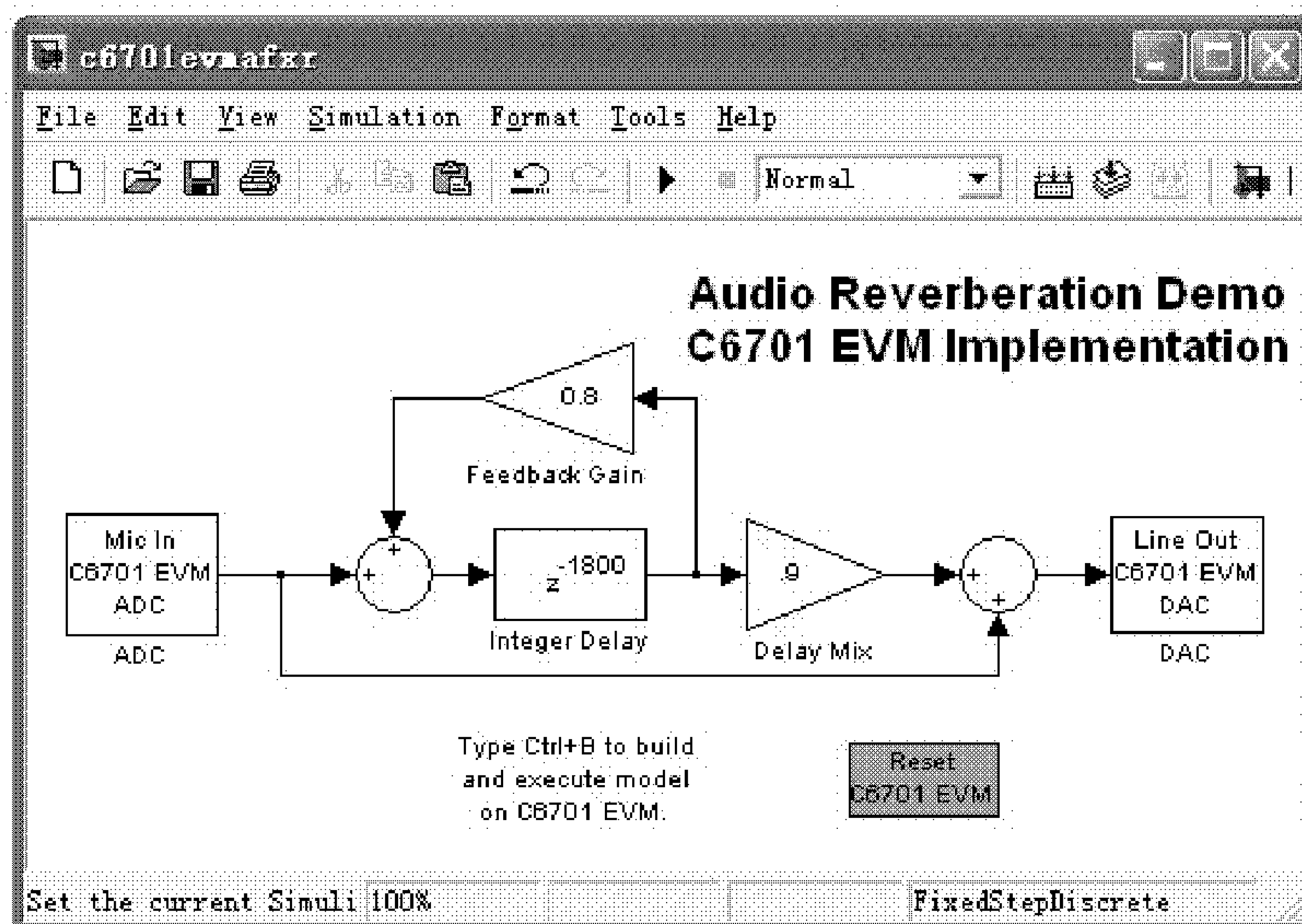


图6.12　c6701evmafxr模型窗口

(2) 选择c6701evmafxr模型窗口→Simulink菜单→Simulation parameter…对话框→Real－Time Workshop面板栏。

(3) 点击Real－Time Workshop面板上的Build按钮，开始代码产生、编译链接、加载并在C6701 EVM目标板上运行c6701evmafxr模型。

MATLAB命令窗会显示过程信息，如果编译链接、加载目标板成功，则最后显示类似如下的信息：

```
C6x EVM Command Line COFF Loader Utility,Version 1.20a
Copyright (c)1998 by DNA Enterprises,Inc.
Found board type:EVM6x Revision:0
Using DSP memory map 1.
###Downloaded:c6701evmafxr.out
###Successful completion of Real-Time Workshop build procedure
for model:c6701evafxr
```

当看到上述信息后，表明c6701evmafxr模型开始在C6701 EVM目标板上运行，接下来可以测试在目标板上的运行情况。如果有出错信息，需要根据错误信息重新配置EVM板和软件。

(4) 对着麦克风说话，就可以从扬声器里听到回声，也可以利用示波器观察输出波形。

(5) 停止C6701 EVM板上程序的执行。

可以利用如下几种方法来停止程序的执行：

- 利用 CCS 中的 Debug→Halt 函数。
- 在 MATLAB 命令窗中输入 halt 命令(MATLAB Link for CCS Development Tools，详见第 5 章)。
- 点击 c6701evmafxr 模型中的 C6701 EVM Reset 模块。

6.7.2　应用 TI C6701 EVM 板的演示例子

本节利用一个简单的例子来演示如何创建模型、编译链接模型以及在 C6701 EVM 板上运行此模型，并观察运行结果。

本节的演示例子与上节的测试模型 c6701evmafxr.mdl 相同，但在本节不是直接应用此模型，而是演示如何一步步创建模型、设置 Simulation Parameters 和 Real－Time Workshop 选项、编译链接模型及在 C6701 EVM 板上运行此模型，并与上节的测试结果进行比较。

1．创建一个模拟声音回音的模型

(1) 打开 Simulink。

在 MATLAB 命令窗中输入 simulink 命令，打开 Simulink 工作窗口。

(2) 选择 Simulink 菜单栏→File→New→Model，打开一个新 Simulink 模型窗口。

(3) 利用 Simulink 模块、DSP Blockset 模块和 C6701 EVM 板支持模块创建如图 6.13 所示的模型。

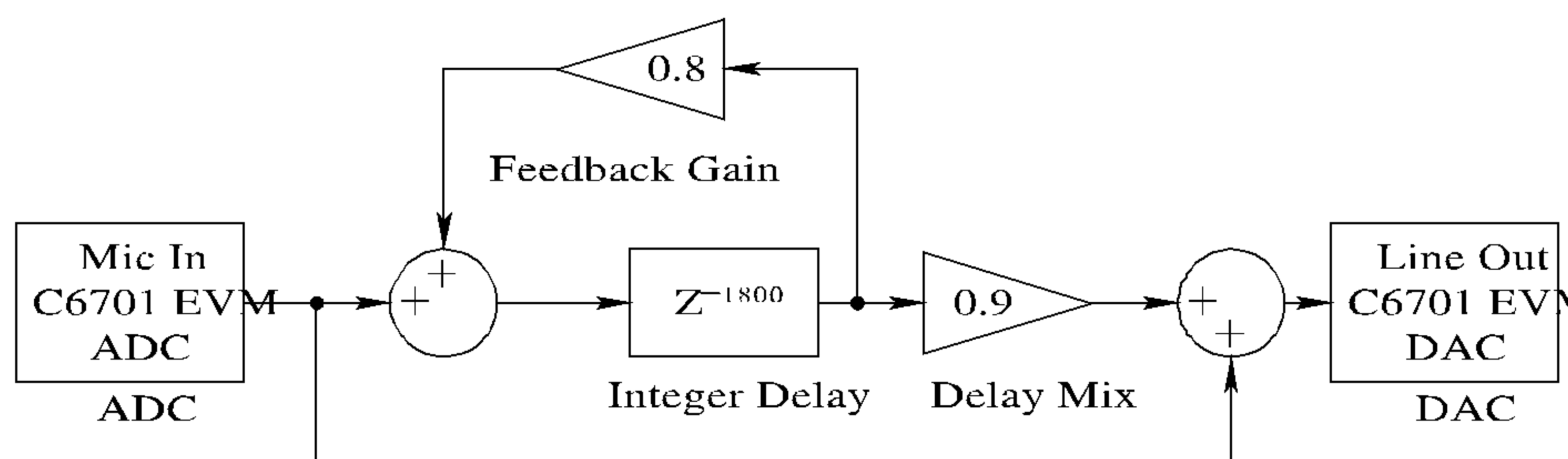


图 6.13　C6701 EVM 板实时 Simulink 模型

(4) 指定路径和模型名保存此模型。

(5) 配置 C6701 EVM 模块参数。

双击 C6701 EVM ADC 模块，打开模块参数对话框，设置参数如下：

- 设置 ADC source 为 Mic In。
- 清除 Stereo 项。
- 选择+20 dB mic gain boost 项。
- 设置 Sample rate 为 8000。
- 设置 Codec data format 为 16－bit linear。
- 设置 Output data type 为 Double。
- 设置 Scaling 为 Normalize。
- 设置 Source gain 为 0.0。
- 在 Samples per frame 项中输入 64。

• 点击 OK，关闭 C6701 EVM ADC 模块参数对话框。

双击 C6701 EVM DAC 模块，打开 C6701 EVM DAC 模块参数对话框，设置参数如下：

• 设置 Codec data format 为 16 - bit linear。
• 设置 Scaling 为 Normalize。
• 设置 DAC attenuation 为 0.0。
• 设置 Overflow mode 为 Saturate。
• 点击 OK，关闭 C6701 EVM DAC 模块参对话框。

2. 配置仿真参数(Simulink Parameters)和 Real - Time Workshop 选项

选择模型窗 → Simulation 菜单 → Simulation parameter…对话框，打开 Simulation parameters 对话框。

(1) 点击 Simulation Parameter 对话框的 Solver 面板栏，在此面板中设置参数如下：

• 设置 Start time 为 0.0，Stop time 为 inf。
• 设置 Solver options 为 Fixed - step 和 discrete。
• 设置 Fixed step size 为 auto，Mode 为 Single Tasking。

(2) 点击 Simulation Parameters 对话框的 Real - Time Workshop 面板栏。

(3) 在 Category 列表中选择 Target configuration，在此面板中设置参数如下：

• 设置 System target file 为 ti_c6000.tlc。

可以通过 Browse 来选择系统目标文件，Real - Time Workshop 会自动配置 Template makefile 和 Make command 项。

(4) 在 Category 列表中选择 TI C6000 target，在此面板中设置 Code generation target type 为 C6701 EVM。

(5) 在 Category 列表中选择 TI C6000 code generation，在此面板中选择 Inline DSP Blockset function。

(6) 在 Category 列表中选择 TI C6000 compiler，在此面板中设置参数如下：

• 设置 Byet order 为 Little_endian。
• 设置 Compiler verbosity 为 Quiet。

(7) 在 Category 列表中选择 TI C6000 linker，在此面板中设置参数如下：

• 选择 Retain .obj files 项。
• 设置 Linker command file 为 Full_ memory_map。

(8) 在 Category 列表中选择 TI C6000 runtime，在此面板中设置参数如下：

• 设置 CPU clock 为 133 MHz。
• 设置 Overrun action 为 Halt。
• 设置 Build action 为 Build_and_execute。

其它的仿真参数和 Real - Time Workshop 选项都利用默认值。

3. 编译链接并在 C6701 EVM 板上运行模型

配置好仿真参数和 Real - Time Workshop 选项后，接下来编译链接、加载和在 C6701 EVM 板上运行此模型。

(1) 点击 Real - Time Workshop 面板上的 Build 按钮，自动完成代码产生、编译链接、

代码加载，并开始在 C6701 EVM 板上运行此模型。

(2) 测试模型在 C6701 EVM 板上的运行情况，对着麦克风讲话，从扬声器里可以听到模拟的回音现象。

(3) 停止 C6701 EVM 板上模型的执行，可以通过如下几种方法来实现：

- 利用 CCS 中的 Debug→Halt 函数。
- 在 MATLAB 命令窗中输入 halt 命令。
- 点击模型中的 C6701 EVM Reset 模块。

6.8 TI C6711 DSK 目标板的应用

TI C6711 DSK 板是 TI 公司推出的一种高性价比的学习板。利用 TI C6711 DSK 板、CCS 和 ETTIC6000 可以进行快速的嵌入式应用开发和硬件在线仿真。

本节介绍如何在 ETTIC6000 环境下开发 TI C6711 DSK 板的实时可执行代码，其中包括配置、验证、测试和开发实时代码的演示例子。

6.8.1 TI C6711 DSK 板的配置、验证和测试

1．配置 TI C6711 DSK 板

按照 CCS 中的帮助说明安装并配置好 TI C6711 DSK 板，在 ETTIC6000 下无须再进行配置。

2．验证 TI C6711 DSK 板的安装

TI 公司提供了一个测试程序，用来验证 DSK 板及其软件是否安装成功。按下列步骤来完成验证：

(1) 打开 DOS 命令窗。

(2) 进入目录\..\ti\C6000\dsk6x11\conftest\。

(3) 在 DOS 命令行中输入 dsk6xtst 命令，开始测试。

(4) 观察测试结果，确定一切工作正常。

如果验证失败，需要重新配置 C6711 DSK 板并重新验证。

3．测试 C6711 DSK 板在 ETTIC6000 环境下是否工作正常

ETTIC6000 提供了几个演示模型，在 MATLAB 命令窗中输入：help tic6000，会显示这些演示模型。其中，c6711dskafxr.mdl 模型用来模拟声音的回音现象，下面利用此模型来测试 C6711 DSK 在 ETTIC6000 环境下的工作情况。

利用 c6711dskafxr 模型进行测试之前，需要把一个麦克风连接到 C6711 DSK 板上的 MIC IN 接头上，把扬声器连接到 C6711 DSK 板上的 MIC OUT 接头上。

按下列步骤完成测试：

(1) 在 MATLAB 命令窗中输入 c6711dskafxr 命令，打开 c6711dskafxr.mdl 模型窗，如图 6.14 所示。

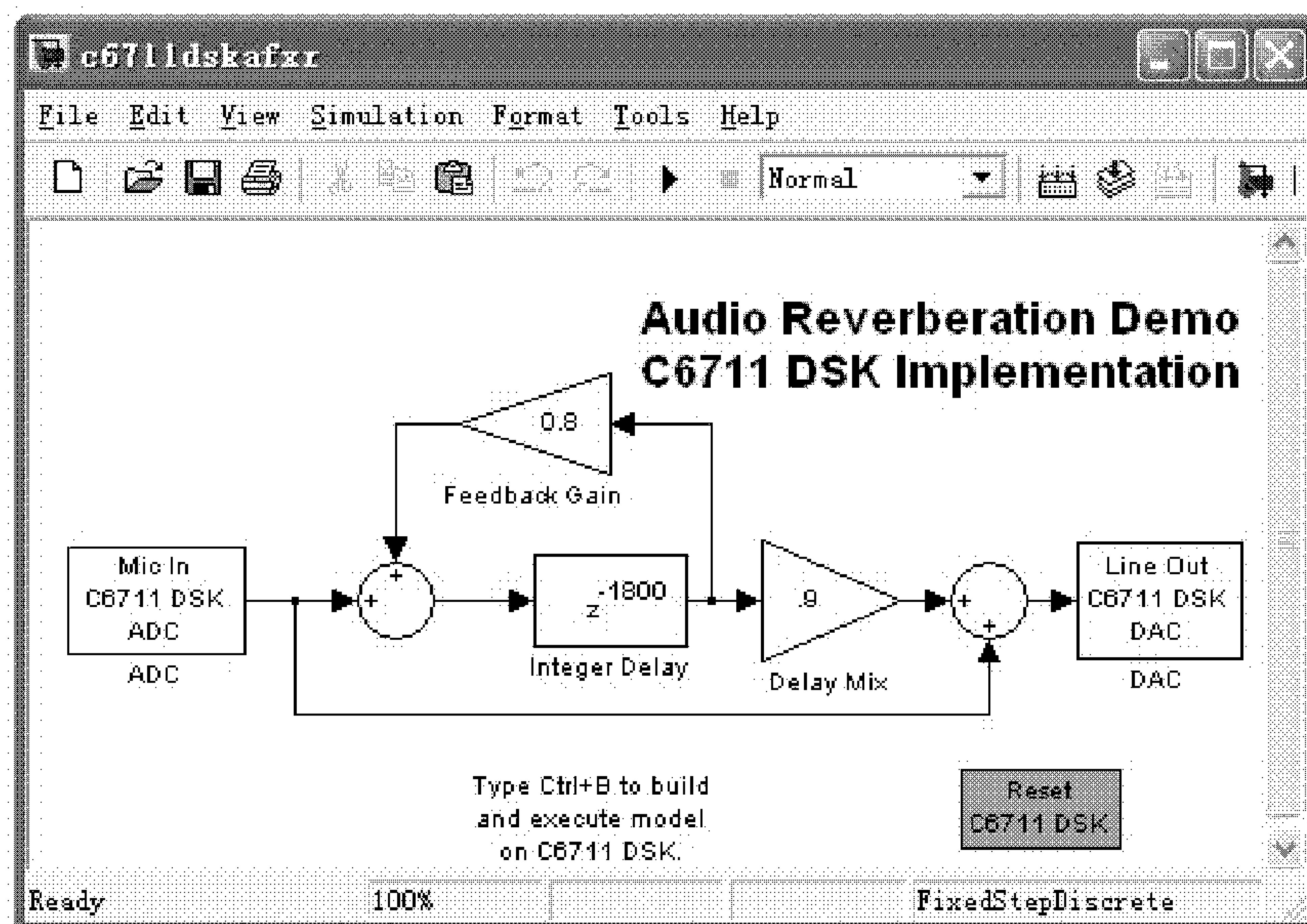


图 6.14　c6711evmafxr 模型窗口

(2) 选择 c6711dskafxr 模型窗→Simulation 菜单→Simulation parameters 对话框→Real－Time Workshop 面板。

(3) 点击 Real－Time Workshop 面板上的 Build 按钮，开始代码产生、编译链接、加载 C6711 DSK 板并开始运行。

如果整个过程运行成功，那么最后显示类似如下的信息：

C6x DSK Command Line COFF Loader Utility,Version 1.20a
Copyright (c)1998 by DNA Enterprises,Inc.
Found board type:DSK6x Revision:0
Using DSP memory map 1.
###Downloaded:c6711dskafxr
###Successful completion of Real-Time Workshop build procedure
for model:c6711dskafxr

当看到上述信息后，表明 c6711dskafxr 模型开始在 C6711 DSK 板上运行。如果看到错误信息，则需要重新配置 DSK 板或软件。

(4) 测试 c6711dskafxr 模型在 C6711 DSK 板上的运行情况，对着麦克风讲话，可以从扬声器里听到模拟的声音回音。

(5) 停止 C6711 DSK 板上 c6711dskafxr 模型的执行，可以利用下述几种方法来实现：

- 选择 CCS IDE 中的 Debug→Halt。
- 在 MATLAB 命令窗中输入 halt 命令。
- 点击模型中的 C6711 DSK Reset 模块。

6.8.2　应用 TI C6711 DSK 板的演示例子

本节演示利用 ETTIC6000 创建嵌入式实时应用的整个过程。

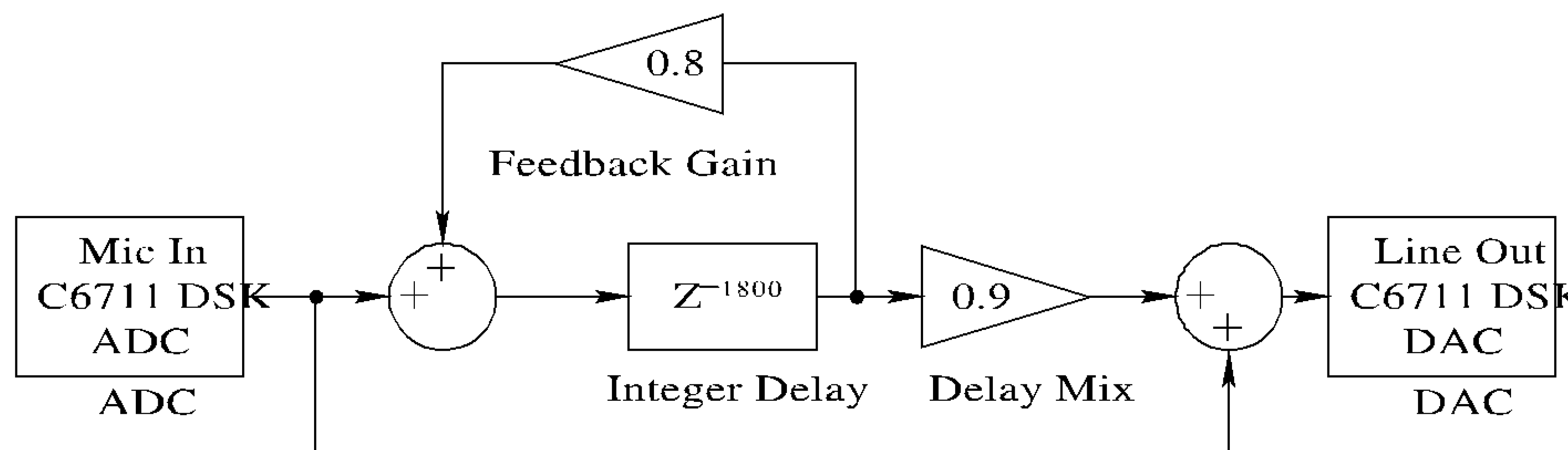
本节演示的实时模型与上节的测试模型相同，但在本节不是直接应用 c6711dskafxr 模型，而是介绍如何一步步创建模型、配置仿真参数、编译链接和加载运行模型。

1．创建一个模拟声音回音现象的模型

(1) 在 MATLAB 命令窗中输入 simulink 命令，打开 Simulink 库窗口。

(2) 选择(Simulink 窗口)File→New→Model，打开一个新的 Simulink 模型窗口。

(3) 利用 Simulink 模块、DSP Blockset 模块和 C6711 DSK 板支持模块，创建一个如图 6.15 所示的模型。



6.15　C6711 EVM 板实时 Simulink 模型

(4) 指定路径和文件名 mytest.mdl，保存此模型。

(5) 配置模块参数。

双击 C6711 DSK ADC 模块，打开模块参数对话框，设置参数如下：

- 设置 ADC source 为 Mic In。
- 选择+20 dB mic gain boost。
- 设置 Output data type 为 Double。
- 设置 Scaling 为 Normalize。
- 设置 Source gain 为 0.0。
- 在 Samples per frame 项中输入 64。
- 点击 OK，关闭掉 C6711 DSK ADC 模块参数对话框。

双击 C6711 DSK DAC 模块，打开模块参数对话框，设置参数如下：

- 设置 Scaling 为 Normalize。
- 设置 DAC attenuation 为 0.0。
- 设置 Overflow mode 为 Saturate。
- 点击 OK 关闭掉此模块参数对话框。

2．设置仿真参数和 Real－Time Workshop 选项

选择模型窗口→Simulation→Simulation parameters…，打开 Simulation parameters: mytest 对话框。

(1) 点击 Simulation parameters:mytest 对话框的 Solver 面板栏，在此面板中设置参数如下：

- 设置 Start time 为 0.0，Stop time 为 inf。
- 设置 Solver options 为 Fixed－step 和 discrete。
- 设置 Fixed step size 为 auto，Mode 为 Single Tasking。

(2) 点击 Simulation Parameters:mytest 对话框的 Real－Time Workshop，打开 Real－Time Workshop 面板。

(3) 从 Category 列表中选择 Target configuration，在此面板中设置 System target file 为 ti_c6000.tlc。

Real－Time Workshop 会自动设置 Template makefile 和 Make command 选项。

(4) 从 Category 列表中选择 TI C6000 target selection，在此面板中设置 Code generation target type 为 C6711 DSK。

(5) 从 Category 列表中选择 TI C6000 compiler，设置此面板的参数如下：

- 设置 Byte order 为 Little_endian。
- 设置 Compiler verbosity 为 Quiet。

(6) 从 Category 列表中选择 TI C6000 linker，设置此面板的参数如下：

- 选择 Retain .obj files 项。
- 设置 Linker command file 为 Full_memory_map。

(7) 从 Category 列表中选择 TI C6000 runtime，设置此面板的参数如下：

- 设置 Overrun action 为 Halt。
- 设置 Build action 为 Build_and_execute。

其它的仿真参数和 Real－Time Workshop 选项都利用默认值。

3．编译链接、加载并在 C6711 DSK 板上运行模型

设置完仿真参数和 Real－Time Workshop 中的选项后，接下来就可以进行编译链接、加载并在 C6711 DSK 板上运行此模型。

(1) 点击 Real－Time Workshop 面板上的 Build 按钮，会自动完成源代码产生、编译链接生成可执行代码、加载并在 C6711 DSK 板上开始运行。

(2) 测试 mytest 模型在 C6711 DSK 板上的运行情况。

对着麦克风讲话，可以从扬声器里听到模拟的回音。

(3) 停止 C6711 DSK 板上 mytest 模型的执行，可以利用以下方法来实现：

- 选择 CCS IDE 中的 Debug→Halt。
- 在 MATLAB 命令窗中输入 halt 命令。
- 点击模型中的 C6711 DSK Reset 模块。

思　考　题

6.1　Embedded Target for the TI TMS320C6000™ DSP Platform 工具的功能是什么？它与第 5 章介绍的 MATLAB Link for CCS Development Tools 工具有何不同？Embedded Target for the TI TMS320C6000™ DSP Platform 工具只支持 TI 公司的哪种类型的 DSP？

6.2　利用 Embedded Target for the TI TMS320C6000™ DSP Platform 和 MATLAB Link for CCS Development Tools 工具是否就可以完成 DSP 软件开发的整个过程，即不再需要传统的编程资源？

6.3 Embedded Target for TI C6000 DSP、Simulink、Real－Time Workshop、MATLAB Link for CCS Development Tools、CCS和目标板之间的关系是什么？为了Embedded Target for TI C6000 DSP能正常工作，其中哪些软件是必需的？

6.4 Real－Time Workshop工具的功能是什么？是否可以利用Real－Time Workshop来生成其它类型DSP的代码？如何做？

6.5 Embedded Target for the TI TMS320C6000™ DSP Platform提供了多个Simulink模块，包括汇编优化的计算模块，用户如何编写自己的汇编语言Simulink模块？

6.6 是否DSP Blockset、Fixed－point Blockset等模块库中的任何模块以及用户利用MATLAB语言自己创建的模块都可用来生成TMS320C6000™ DSP的可执行代码？生成代码的执行效率和长度如何？

6.7 用户能否把Simulink模型生成其它类型的DSP的可执行代码(例如，SHARC DSP)？如果要这么做，用户需要进行哪些开发？

6.8 利用Embedded Target for the TI TMS320C6000™ DSP Platform中提供的Simulink模型例子来演示整个TMS320C6000™ DSP可执行代码的生成和调试过程。

第 7 章　直接由 Simulink 模型生成 SHARC DSP 的目标代码

类似于第 6 章中介绍的直接由 Simulink 模型生成 TMS320C6000 DSP 的目标代码的内容，本章介绍如何从 Simulink 模型生成 SHARC DSP 的目标代码。在 MATLAB 下开发两种 DSP 可执行代码的思路及操作步骤也非常类似，本章只介绍从 Simulink 模型生成 SHARC DSP 目标代码的主要操作过程。感兴趣的读者可以登陆到 www.sdltd.com/dspdeveloper，以得到更多的详细资料。

需要说明的是，第 6 章中介绍的 Embedded Target for the TI TMS320C6000™ DSP Platform 产品是由 TI 公司和 Mathswork 公司联合开发的，并集成在 MATLAB6.5(R13)或更高版本中。而本章中介绍的 DSPdeveloper 产品是 SDL 公司开发的，因此用户需要另外从 SDL 公司购买此产品，读者也可以从 www.sdltd.com/dspdeveloper 网站上得到一个免费试用版的注册码及其软件。

类似于第 6 章的 Embedded Target for the TI TMS320C6000™ DSP Platform 工具，SDL 公司开发的 DSPdeveloper 也需要与 MATLAB 中的 Simulink、Real－Time Workshop 工具以及 AD 公司的开发环境 VisualDSP++配合使用(而 Embedded Target for the TI TMS320C6000™ DSP Platform 需要与 CCS 配合)，从而形成一个系统级的集成环境。在此环境下就可以完成目标代码开发的整个过程，该过程包括概念设计、仿真、代码生成、调试和运行。

7.1　DSPdeveloper 概述

如前所述，SDL 公司开发的 DSPdeveloper 需要与 MATLAB 中的 Simulink、Real－Time Workshop 工具以及 AD 公司的 DSP 开发环境 VisualDSP++配合使用，从而形成一个系统级集成环境，在此环境下就可以完成目标代码开发的整个过程。如果在目标代码中集成 VCSE(Visual Component Software Engineering)技术，则还需要用到 MATLAB 中的 Real－Time Workshop Embedded Coder 工具。DSPdeveloper 还提供了多种 Simulink 模块，包括支持 ADSP21065L EZ－KIT Lite、ADSP21160 EZ－KIT Lite、ADSP21161 EZ－KIT Lite 和 LanSHARC 目标板的模块以及汇编语言编写的 FIR、IIR 滤波器模块等。开发人员在 Simulink 环境中利用 MATLAB 本身提供的模块和 DSPdeveloper 提供的模块来构造实时模型，并进行功能模拟。当模拟结果满意后，就可以直接把 Simulink 模型生成 SHARC DSP 的目标代码。整个代码生成过程都是自动完成的，DSPdeveloper 首先调用 Real－Time Workshop 工具把 Simulink 模型生成 C 程序，然后再利用 AD 公司的开发环境 VisualDSP++

把 C 程序编译链接，生成 SHARC DSP 的目标代码。

DSPdeveloper、Simulink、Real - Time Workshop、VisualDSP++ 和调试目标之间的关系如图 7.1 所示。

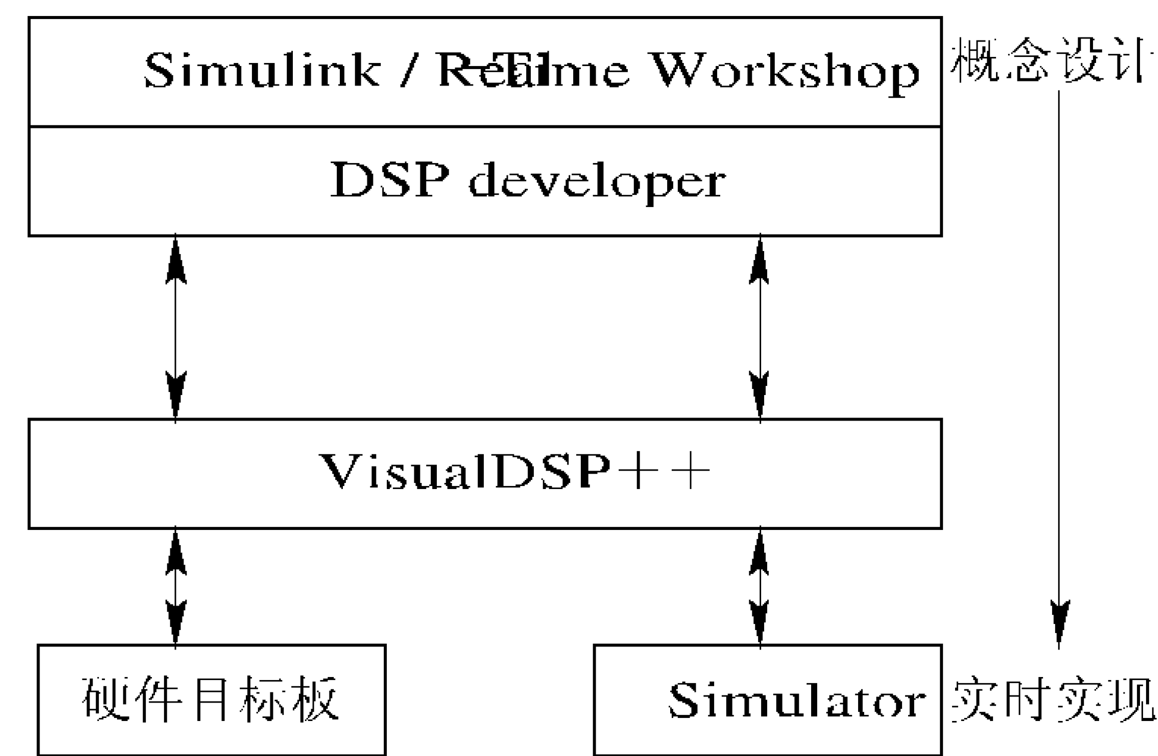


图 7.1 DSPdeveloper 与其它所需产品之间的关系

如上所述，DSPdeveloper 还需要与其它软、硬件配合使用才能正常工作。以 DSPdeveloper 的 1.3 版本为例，需要配合的软件包括 MATLAB6.1、Simulink4.1、Real - Time Workshop4.1、Real - Time Workshop Embedded Coder2.0(应用 VSCE 功能时才需要)、VisualDSP++2.0 for SHARC。DSPdeveloper 要求必须先安装好 MATLAB 和 VisualDSP++ 后，再安装 DSPdeveloper。安装 DSPdeveloper 过程中必须指定好 MATLAB 的安装路径。DSPdeveloper 对配合软件的版本要求非常严格，因此用户必须首先确定 DSPdeveloper 指定的配合软件的版本号，然后安装这些相应软件，否则编译链接过程中会出错。DSPdeveloper 可以支持多种硬件目标板和软件模拟器(Simulator)，硬件目标板可以是 ADSP 21065L EZ - KIT、ADSP 21160 EZ - KIT、ADSP 21161N EZ - KIT、LanSharc 以及用户自制的目标板。

7.2 DSPdeveloper 提供的模块

在 MATLAB 命令窗中输入 dspdeveloper，就会打开 DSPdeveloper 提供的 Simulink 模块库和模块，如图 7.2 所示。利用 Simulink 本身提供的模块和 DSPdeveloper 提供的模块来构造实时模型，DSPdeveloper 几乎可以把 Simulink 中构造的任何模型都生成 SHARC DSP 的可执行代码。

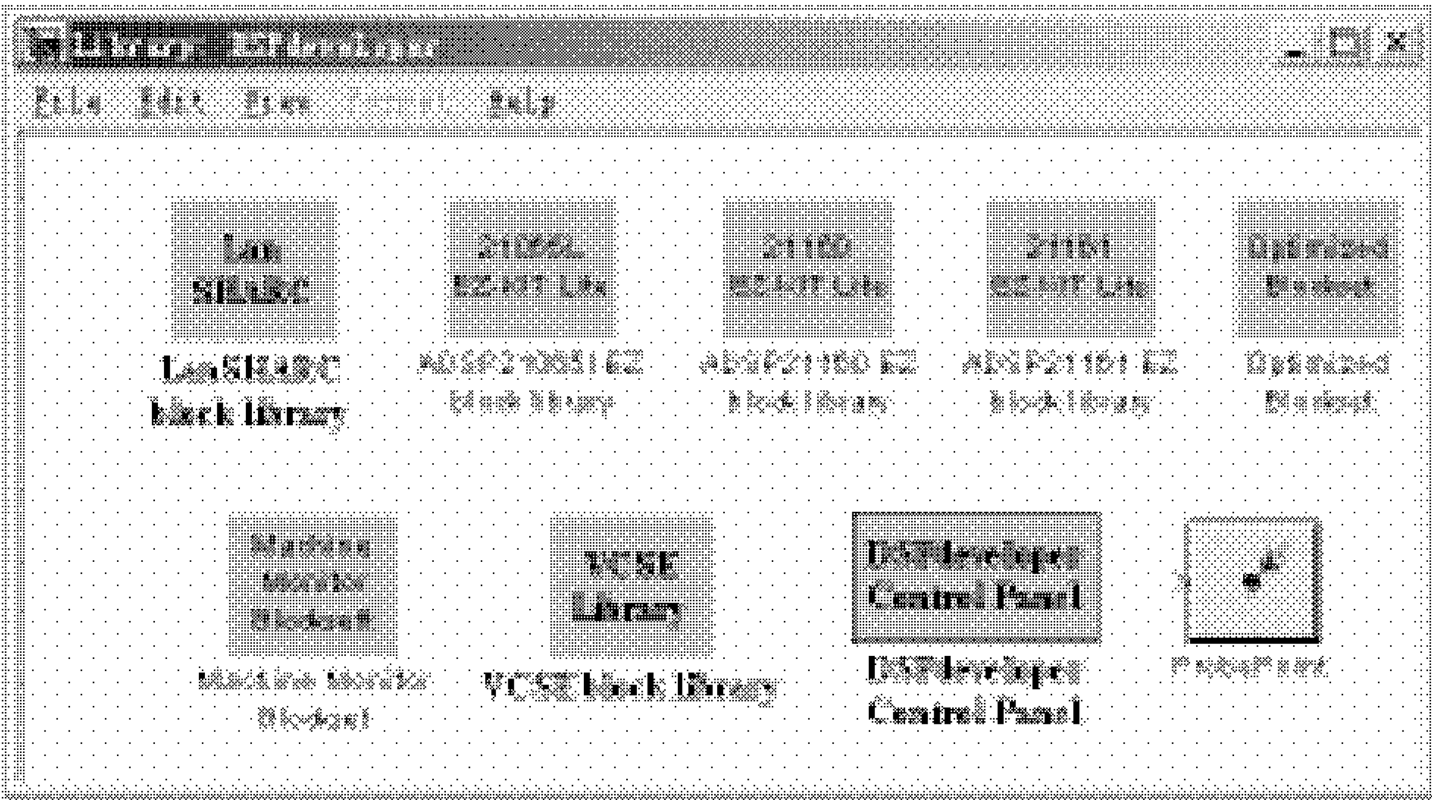


图 7.2 DSPdeveloper 提供的 Simulink 模块

• DSPdeveloper Control Panel 模块：此模块可以添加到任何 Simulink 模型中，且不需要与任何模块进行连接。DSPdeveloper Control Panel 是 DSPdeveloper 的主控面板，其功能是用来选择调试目标板或软件模拟器、复位 DSP、加载目标代码并运行、控制主机与目标 DSP 之间的通信以及设置探点(ProbePoint)等。

• ProbePoint 模块：此模块在目标 DSP 中保存数据，从而可利用 DSPdeveloper Control Panel 来获取此数据并进行观察。

• Optimized Blockset 模块库：包含优化的汇编语言编写的 FIR 滤波器、IIR 滤波器和基 4 滤波器模块。

• Machine Monitor Blockset 模块库：包含多个监控和报警模块。

• 21160 EZ－KIT Lite 模块库：包含专门用于支持 ADSP21160 EZ－KIT Lite 目标板的模块。

• 21161 EZ－KIT Lite 模块库:包含专门用于支持 ADSP21161 EZ－KIT Lite 目标板的模块。

• 21065L EZ－KIT Lite 模块库：包含专门用于支持 ADSP21065L EZ－KIT Lite 目标板的模块。

• LanSHARC 模块库：包含专门用于支持 LanSHARC 目标板的模块。

• VCSE 模块库：包含支持 VCSE 技术开发的模块。

Simulink中的几乎任何内在模块都可用于创建目标模型，但下面的一些与主机接口的内在模块不能用于DSPdeveloper：Scope、Display、XY Graphs、To File、From File、To Workspace、From Workspace、MATLAB Fnc、Algebraic Constraint。

DSPdeveloper还提供了多个Simulink模型演示例子，保存在DSPdeveloper安装路径\Sharc \ Examples\目录下。

7.3　应用 DSPdeveloper 进行实时代码开发的步骤

应用 DSPdeveloper 及其配合软件开发 SHARC DSP 目标代码的主要操作步骤如下：

(1) 在 MATLAB 命令窗中输入 simulink 命令，打开 simulink 库窗口。

(2) 在 Simulink 环境中利用 Simulink 的内在模块和 DSPdeveloper 提供的模块来创建实时模型，图 7.3 为 DSPdeveloper 提供的一个 Simulink 演示模型。

在图 7.3 的 Simulink 模型例子中，两个信号源模块(输出正弦波)都是 Simulink 的内在模块，而其它模块都是 DSPdeveloper 提供的，包括 AD1881 Codec 模块(支持 ADSP21160 EZ－KIT Lite 目标板的模块)、探点(ProbePoint)模块、DSPdeveloper 控制面板(DSPdeveloper Control Panel)模块。

(3) 创建完成 Simulink 模型后，还需要在 Simulink 环境中对此模型进行仿真。只有当此模型在 Simulink 中仿真通过后，才能把它生成目标 DSP 代码。

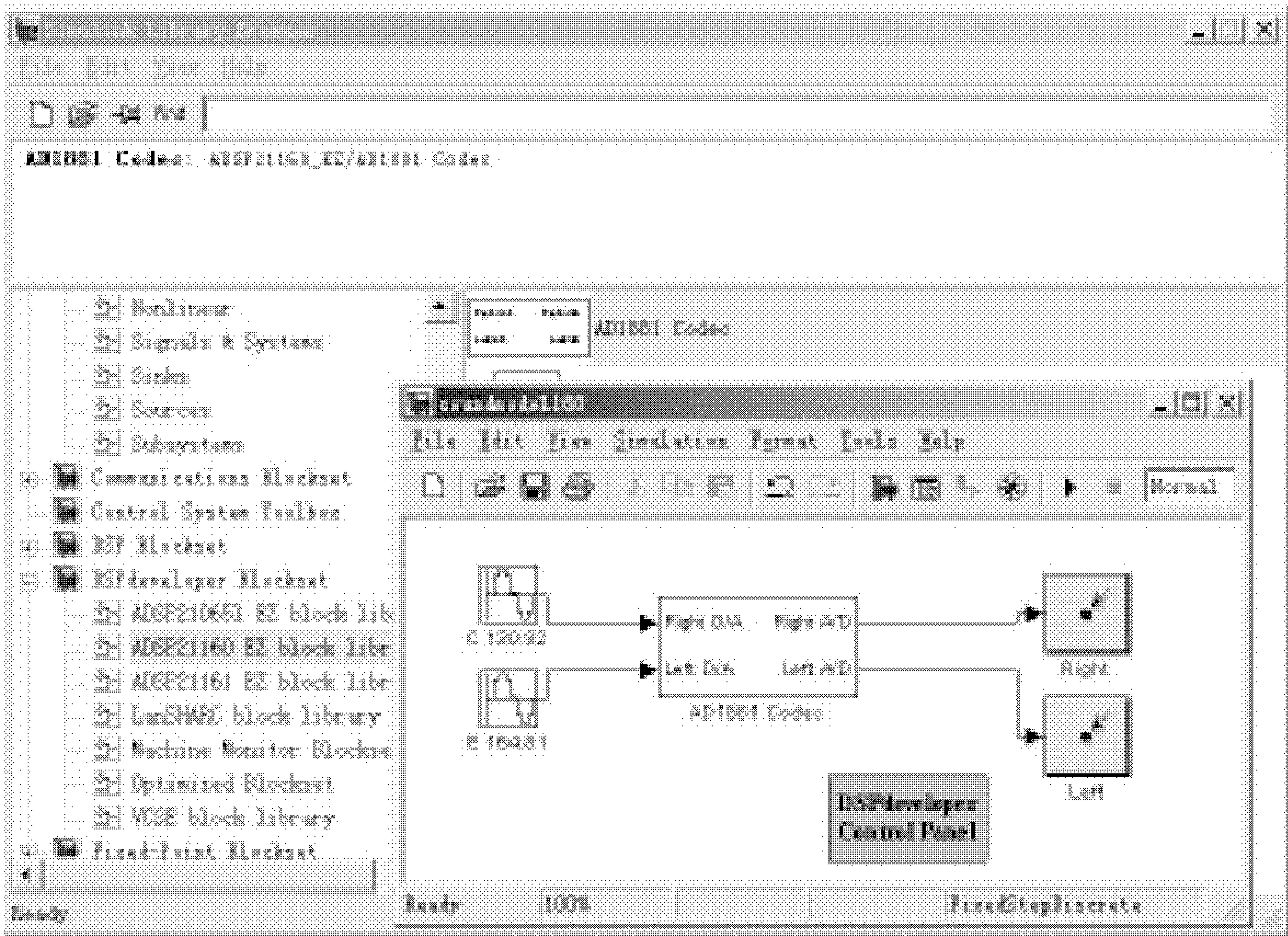


图 7.3　Simulink 演示模型

(4) 在把 Simulink 模型生成目标代码之前，还必须首先设置 Real－Time Workshop 选项，即指定 Real－Time Workshop 如何把 Simulink 模型生成 DSP 的可执行代码。在 Simulink 模型窗中选择 Tools→Real－Time Workshop→Options，打开 Real－Time Workshop 设置面板，如图 7.4 所示。

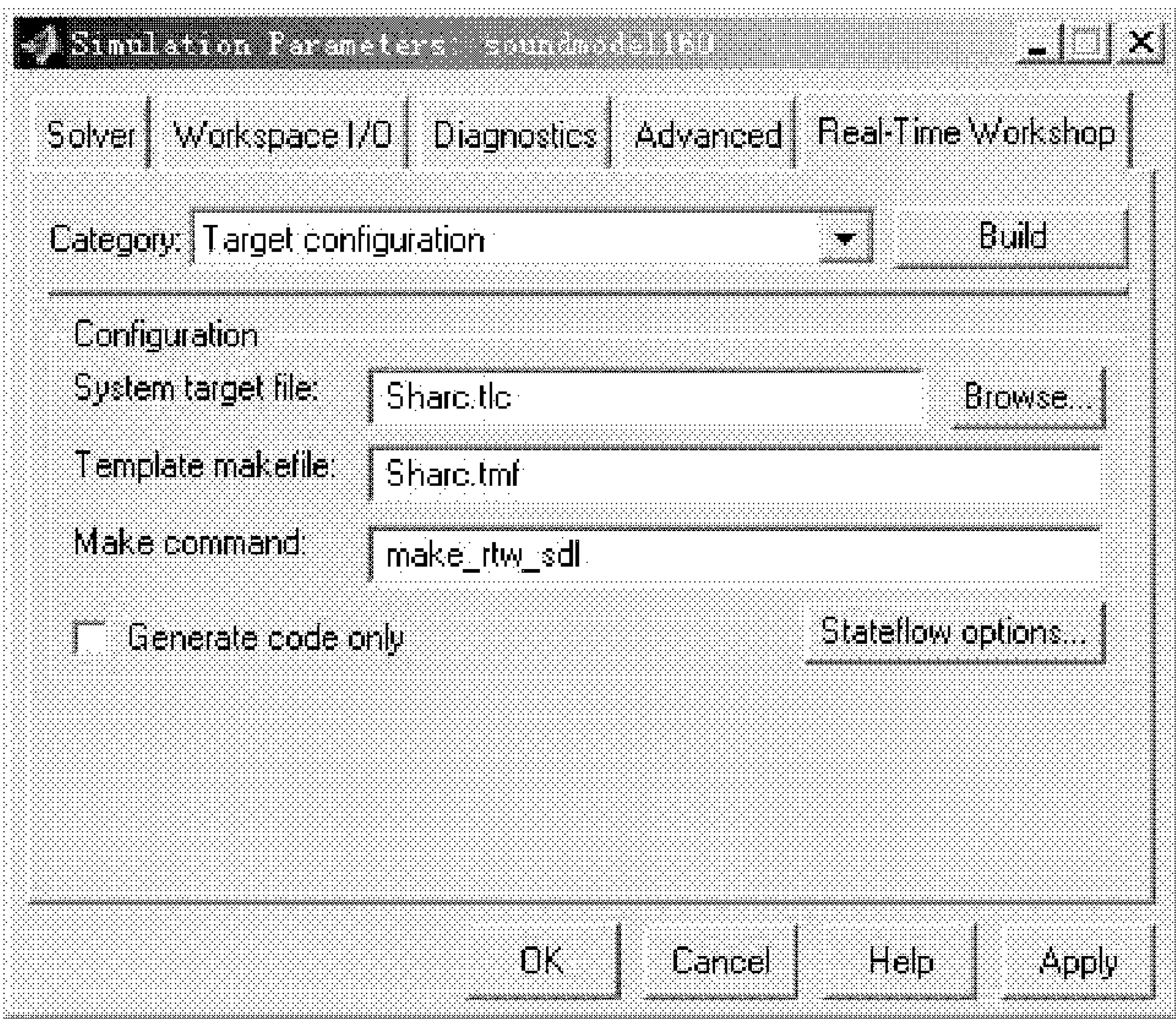


图 7.4　Real－Time Workshop 的设置面板

在 Real－Time Workshop 面板的 Category 项中选择 Target Configuration，点击 System Target file 选项右边的 Browse 按钮，打开 System Target file Browser 窗口，在系统目标文件列表中选择 Sharc.tlc。指定好 System Target file 项后，Template makefile 和 Make command

项会自动配置。

在 Real－Time Workshop 面板的 Category 项中选择 SHARC code generation options，在此面板中选择目标 DSP 的类型(Target Processor 项)以及目标板(Target Hardware 项)，并指定相应的链接描述文件(LDF)。

关于 Real－Time Workshop 面板的其它选项设置，读者可以查看第 6 章中的介绍。

打开 Solver 面板，在此面板中的 type 项中选择 Fixed－Step、discrete，在 Stop Time 项中输入 0.0。

(5) 设置好 Real－Time Workshop 中的选项后，接下来就可以把 Simulink 模型生成目标 DSP 的可执行代码了。点击 Real－Time Workshop 面板中的 Build，或选择 Tools→Real－Time Workshop→Build Model，开始自动编译、链接模型，最终生成 SHARC DSP 的可执行代码(.dxe)。MATLAB 命令窗中会显示此编译链接过程信息，如图 7.5 所示。

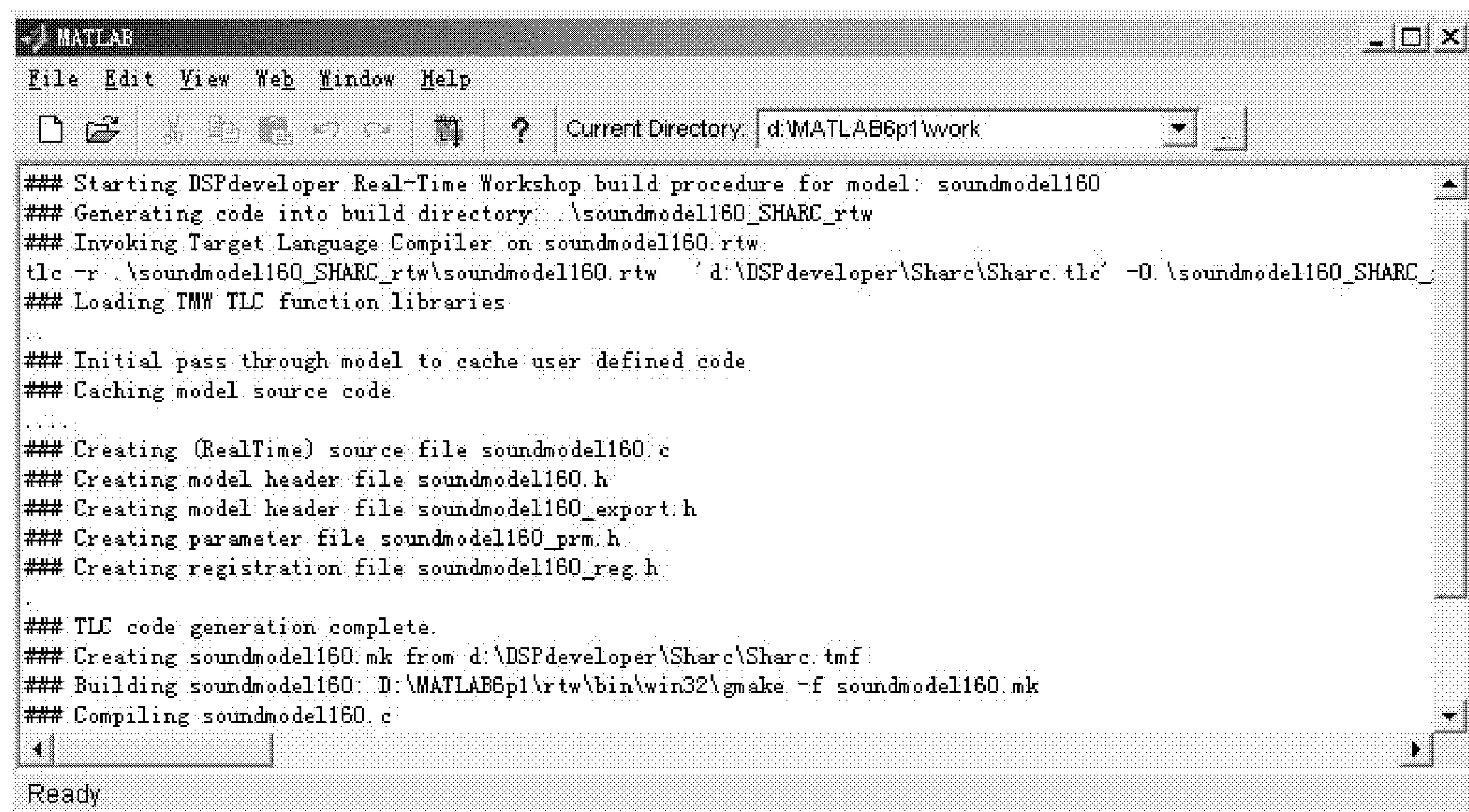


图 7.5　MATLAB 命令窗中显示的编译链接过程信息

(6) 利用 DSPdeveloper Control Panel 控制面板或 VisualDSP++把生成的可执行代码加载到目标板或软件模拟器中进行调试和测试。

接下来介绍如何利用 DSPdeveloper 提供的控制面板(DSPdeveloper Control Panel)和探点(ProbePoint)来加载、运行和验证生成的可执行代码。我们在前面提到过，DSPdeveloper Control Panel 是 DSPdeveloper 提供的一个主控制面板，它具有友好的图形用户接口，可以用来选择加载目标板或 Simulator，加载和运行可执行代码，控制主机与目标 DSP 之间的通信以及获取并显示探点的数据等。探点用来在目标 DSP 的存储器中保存数据。下面假定 Simulink 模型中已经加入了 DSPdeveloper Control Panel 模块和 Probe Point 模块(如图 7.3 中的模型)。

(7) 双击 Simulink 模型中的 DSPdeveloper Control Panel 模块，打开 DSPdeveloper 控制面板，如图 7.6 所示。

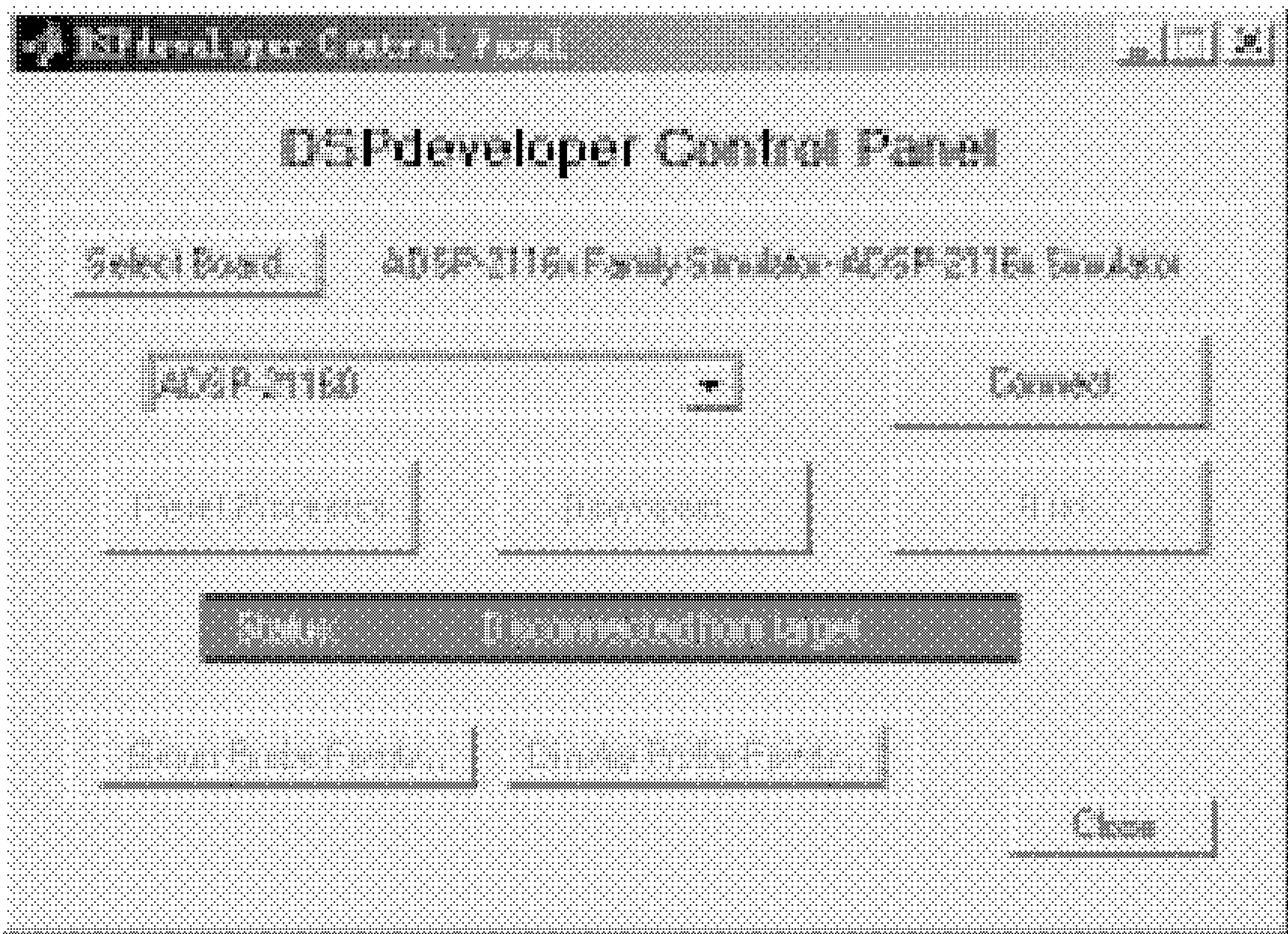


图 7.6　DSPdeveloper 控制面板

- Select Board：选择需要连接并加载的目标板或软件模拟器。点击 Select Board，会打开一个目标板选择菜单，在此菜单中选择所需的目标板。
- Processor Dropdown List：对于装有多个 DSP 的目标板，此下拉菜单允许用户选择指定的处理器。
- Connect：开始主机(PC)与目标 DSP 之间的通信。
- Reset Processor：复位目标板。
- Download：把 DSPdeveloper 生成的可执行代码(.dxe)加载到目标板中。
- Run：运行目标 DSP 中已加载的可执行代码。
- Status Bar：显示目标板的当前状态。
- Setup Probe Points：设置探点。点击此项，会打开一个 DSPdeveloper 探点设置菜单。
- Display Probe Points：图形显示探点中的数据。

(8) 点击 DSPdeveloper 控制面板上的 Select Board 按钮，打开目标板列表(如图 7.7 所示)，从中选择用于加载调试的目标板或 Simulator。

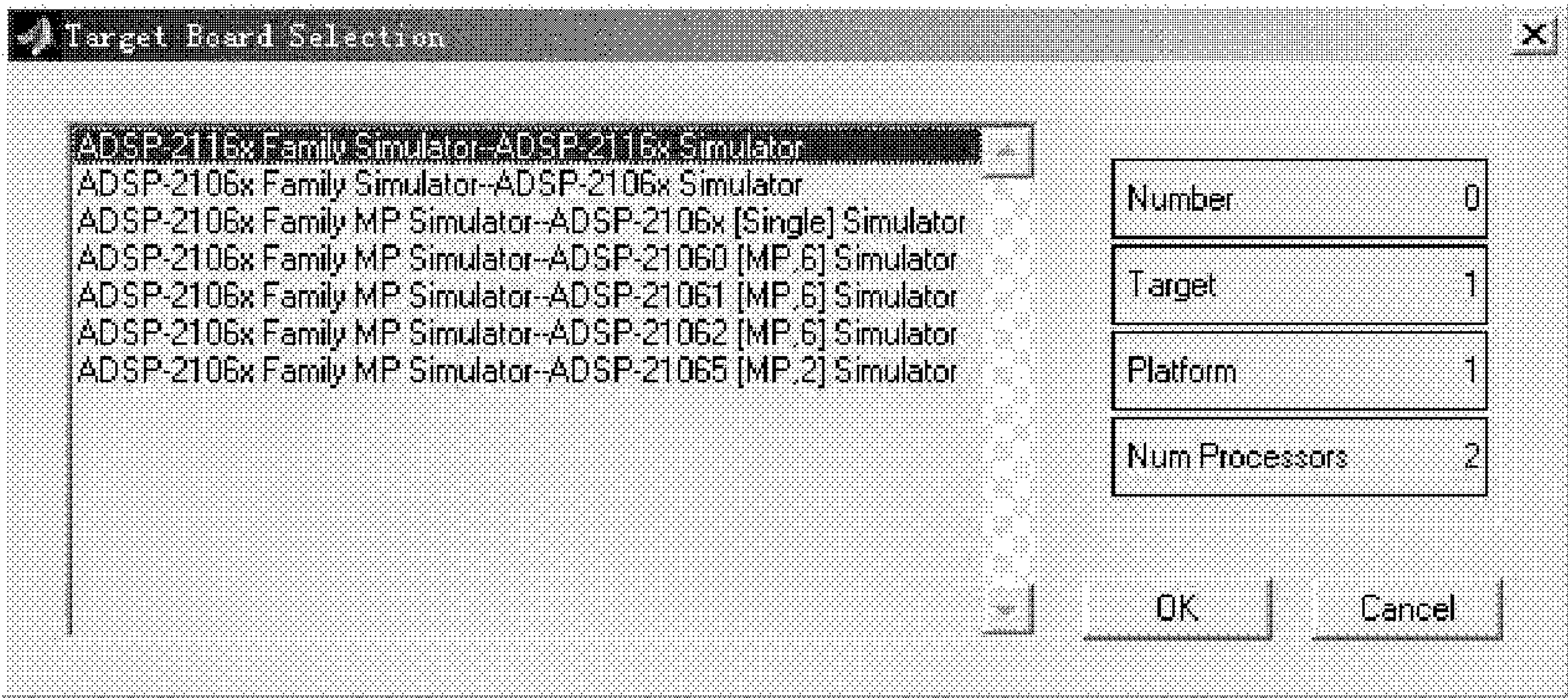


图 7.7　调试目标板列表

(9) 选择好调试目标板后，再点击 DSPdeveloper 控制面板上的 Connect 按钮，开始主机与目标板之间的通信。Status 栏中会显示连接成功信息。

(10) 向目标板加载程序之前，还需要先复位目标板。点击 Download 按钮，选择生成的可执行文件(.dxe)并打开，把可执行文件加载到目标板中。

(11) 设置探点，用图形显示探点收集的运行结果。点击 DSPdeveloper 控制面板上的 Setup Probe Points 按钮，打开探点设置对话框(如图 7.8 所示)，在此对话框中设置探点。

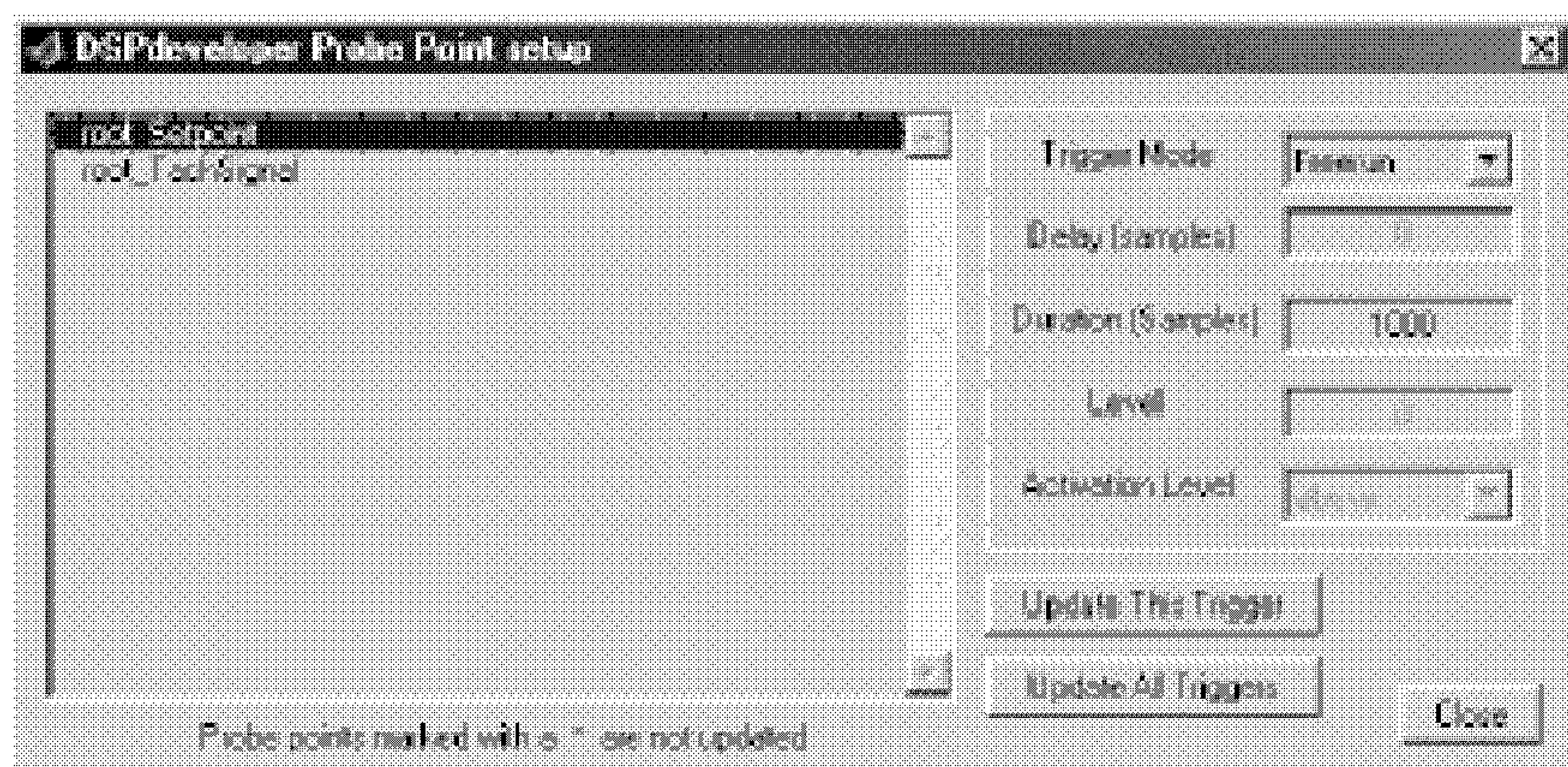


图 7.8　探点设置对话框

• Trigger Mode：选择探点的运行模式，有以下三种模式可供选择：Disabled(关闭探点，不再返回任何数据)、Freerun(返回探点接收到的所有数据)、Triggered(当某一触发条件满足时才返回数据)。

• Delay(samples)：当触发条件满足后还要等待的采样数。

• Duration(samples)：当触发后探点收集的采样数。

• Level：触发电平。

• Activation Level：选择触发条件当数据超过(Above)或低于(Below)触发电平时，触发发生。

• Updata This Trigger：把修改应用于当前选择的探点。Updata All Triggers 把修改应用于所有探点。

(12) 设置完探点后，点击 DSPdeveloper 控制面板上的 Display Probe Points 按钮，打开探点数据显示窗口，如图 7.9 所示。

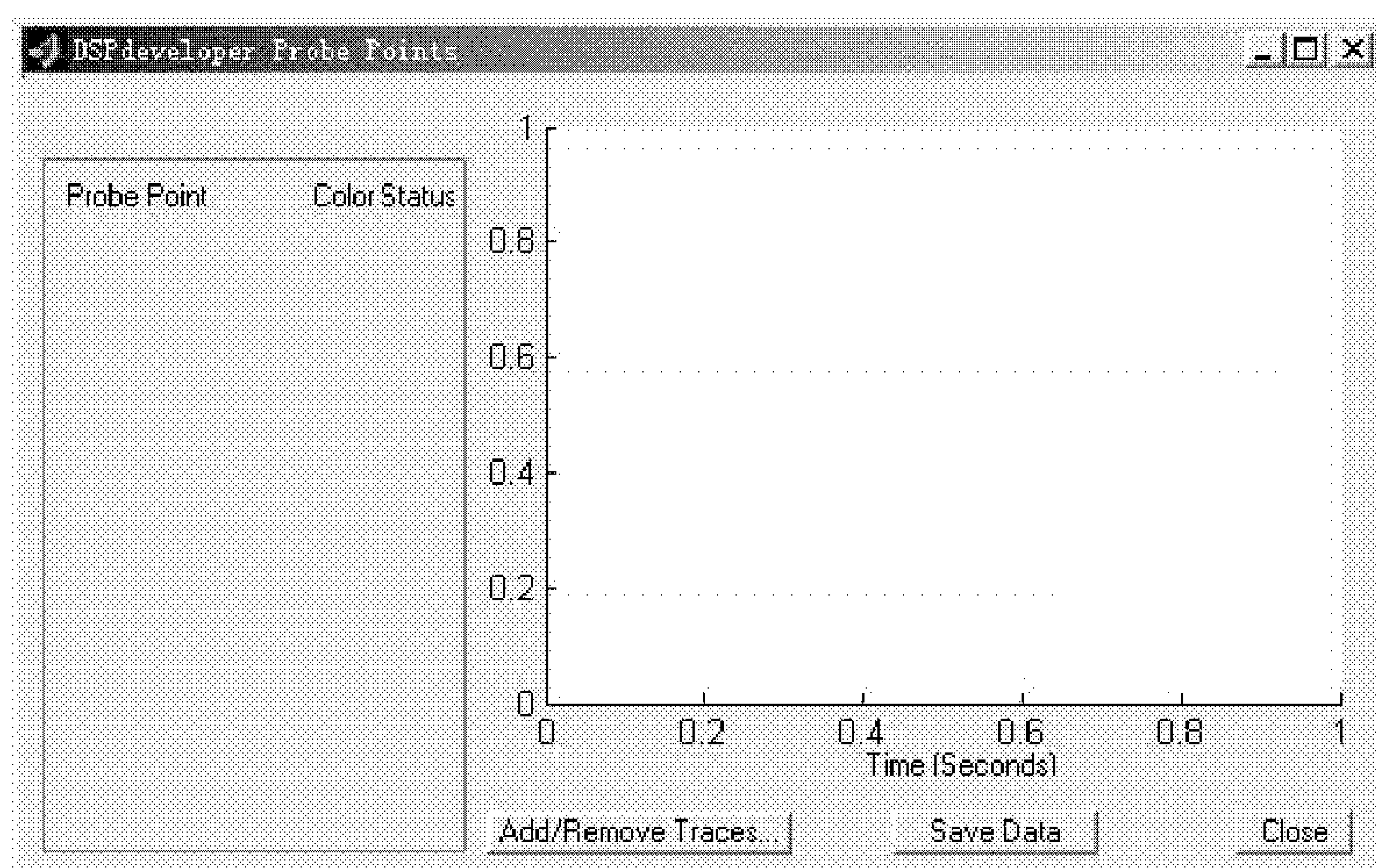


图 7.9　探点数据显示窗口

- Add/Remove Traces：选择在图形窗中需要显示的探点。
- Save Data：把图形中显示的数据保存到主机文件中。

(13) 点击 Run 按钮，开始运行已加载的目标代码。探点数据显示窗口中会显示探点获取的数据。

(14) 根据实际运行结果，在 Simulink 中重新修改模型、编译链接并加载运行，直到实际的运行结果满足所有要求时为止。

7.4　应用 DSPdeveloper 进行实时代码开发的演示例子

在 7.3 节中我们已经详细介绍了从 Simulink 模型生成 SHARC DSP 目标代码的主要过程，本节再利用一个简单的 Simulink 模型来演示上述过程，以帮助读者进一步熟悉 DSPdeveloper 及其操作过程，快速开发出自己的实时模型。

1. 创建 Simulink 模型

在 MATLAB 命令窗中输入 simulink，打开 Simulink 模块窗，利用 Simulink 的内在模块和 DSPdeveloper 提供的模块创建实时模型，并设置每一模块的参数，如图 7.10 所示。

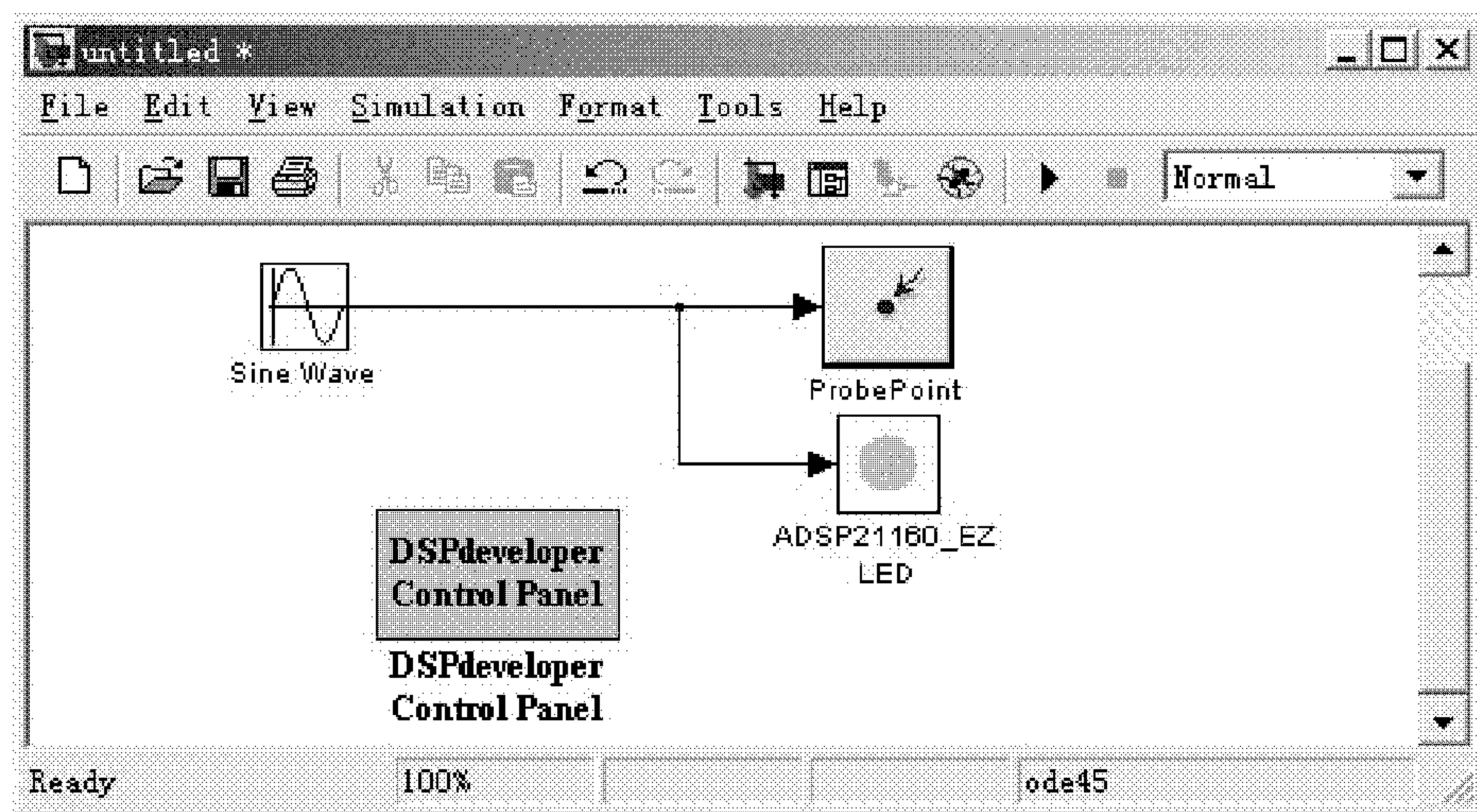


图 7.10　在 Simulink 中创建的实时模型

设置 Sine Wave 模块的参数如下：

- Frequency 为 2*pi*10。
- Sample time 为 1/100。

其它模块参数可利用默认值。

2. 设置仿真参数和 Real－Time Workshop 选项

选择模型窗→Simulation 菜单→Simulation parameter…对话框，打开 Simulation parameters 对话框。

(1) 点击 Simulation Parameter 对话框的 Solver 面板栏，在此面板中设置参数如下：

- 设置 Start time 为 0.0，Stop time 为 5.0。
- 设置 Solver options 为 Fixed - step 和 discrete。
- 设置 Fixed step size 为 auto，Mode 为 auto。

(2) 点击 Simulation Parameters 对话框的 Real - Time Workshop 面板栏。

(3) 在 Category 列表中选择 Target configuration，设置 System target file 为 Sharc.tlc。

可以通过 Browse 来选择系统目标文件。Real - Time Workshop 会自动配置 Template makefile 和 Make command 项。

(4) 在 Category 列表中选择 SHARC code generation options，在此面板中设置参数如下：

- 设置 Clock source for mode 为 Internal - Timer。
- 设置 Target Processor 为 ADSP - 21160。
- 设置 Target Hardware 为 EZ21160。
- 设置 LDF File 为 DSPdeveloper 安装目录\Sharc\Ez21160\ADSP21160_EZ_demo.ldf 或 ADSP21160_EZ_full.ldf。

其它的仿真参数和 Real - Time Workshop 选项都使用默认值。

3．编译链接 Simulink 模型，生成 SHARC DSP 的可执行代码

选择 Tools→Real - Time Workshop→Build Model，开始自动编译、链接模型，最终生成 SHARC DSP 的可执行文件 *modelname*.dxe，DSPdeveloper 会自动把生成的可执行文件放入 *modelname*_SHARC_rtw 目录中(*modelname* 为模型名)。MATLAB 命令窗中会显示此编译链接过程信息。

4．把生成的可执行代码加载到目标板中

利用 DSPdeveloper 控制面板来加载并运行生成的可执行代码。双击模型窗中的 DSPdeveloper Control Panel 模块，打开 DSPdeveloper 控制面板。

(1) 利用 Select Board 按钮来选择 EZ - KIT Lite [ADSP - 21160] 目标板。

(2) 点击 Connect 按钮，建立主机与目标板之间的通信。

(3) 点击 Download 按钮，选择前面生成的可执行文件并把它加载到目标板中。

5．运行已加载的可执行代码并观察探点数据

(1) 点击 Setup Probe Points 按钮，打开探点设置对话框，在此对话框中配置如下参数：

- Trigger Mode 选择为 Freerun。
- Duration 设置为 1000。
- Delay 和 Level 都设置为 0。
- Activation Level 选择为 above。

(2) 点击 Display Probe Points 按钮，打开探点显示窗口，再利用 Add/Remove Traces 按钮，向显示窗口中添加探点。

(3) 最后点击 DSPdeveloper 控制面板上 Run 按钮，开始运行目标 DSP 中的程序。当程序运行结束后，探点显示窗口的显示结果如图 7.11 所示。

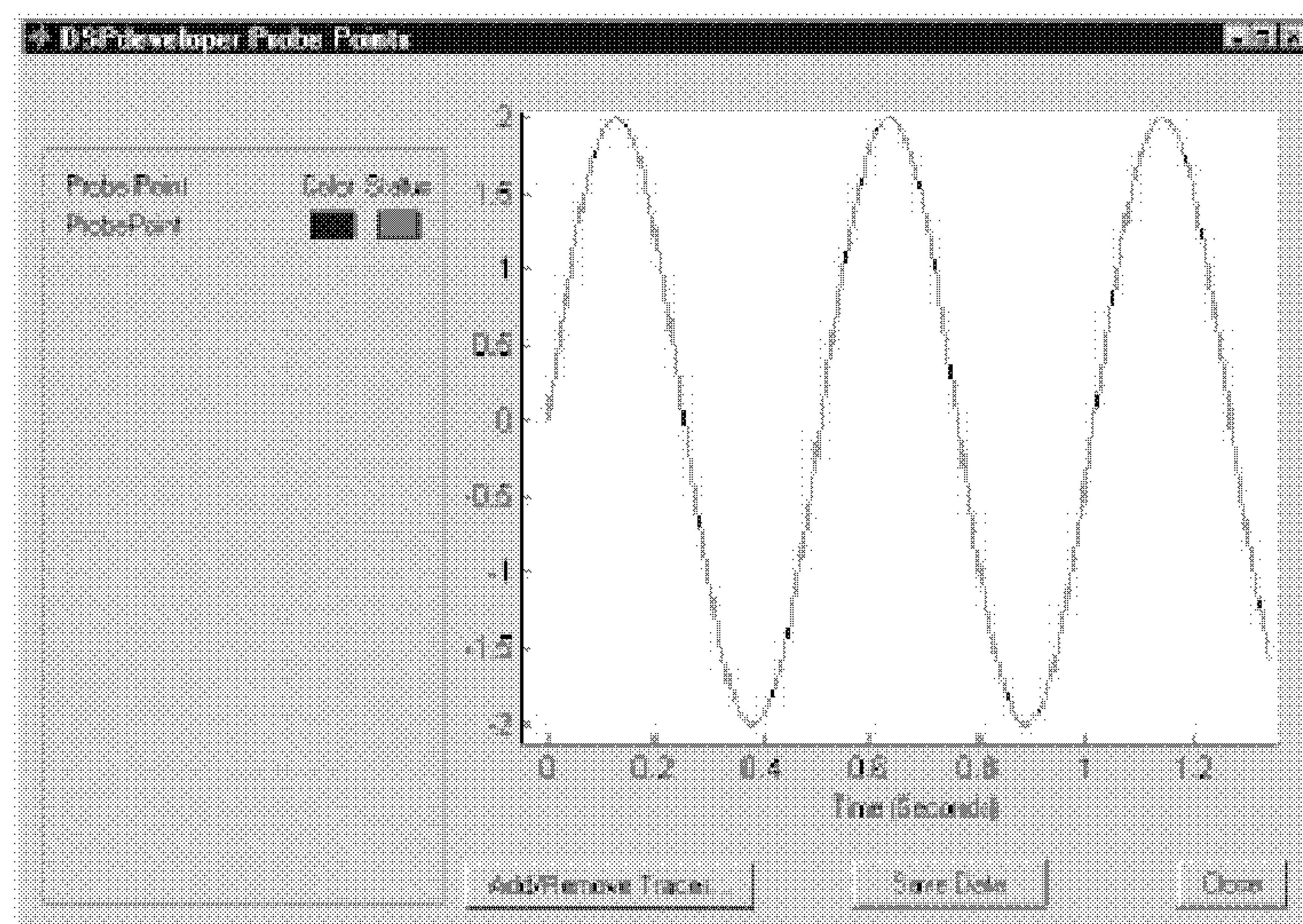


图 7.11　探点数据显示结果

思　考　题

7.1　DSPdeveloper for SHARC 工具的功能是什么？DSPdeveloper for SHARC 工具是哪个公司开发的？可以支持 AD 公司的哪些类型的 DSP？它与第 6 章介绍的 Embedded Target for the TI TMS320C6000™ DSP Platform 工具的功能是否相同？

7.2　DSPdeveloper、Simulink、Real - Time Workshop、VisualDSP++和调试目标之间的关系是什么？

7.3　本章介绍如何把 Simulink 模型直接生成 SHARC DSP 的可执行代码，而在第 6 章中介绍如何把 Simulink 模型直接生成 TMS320C6000™ DSP 的可执行代码，它们的操作步骤及原理是否相同？

7.4　用户能否在 DSPdeveloper 工具的基础上进行补充，以进一步完善其功能，并且提供更多汇编优化的模块？

7.5　利用 DSPdeveloper 中提供的 Simulink 模型例子来演示整个 SHARC DSP 可执行代码的生成和调试过程。

参 考 文 献

1 TMS320C54x DSP Reference Set. Volume 1: CPU and Peripherals. TEXAS Instruments. 2001
2 TMS320C54x Assembly Language Tools User's Guide. TEXAS Instruments. 2002
3 TMS320C54x DSP Programmer's Guide. TEXAS Instruments. 2001
4 TMS320VC5501 Fixed－Point Digital Signal Processor. TEXAS Instruments. 2003
5 TMS320VC5502 Fixed－Point Digital Signal Processor. TEXAS Instruments. 2003
6 TMS320C6000 CPU and Instruction Set Reference Guide. TEXAS Instruments. 2000
7 TMS320C6000 Peripherals Reference Guide. TEXAS Instruments. 2001
8 TMS320C6000 Assembly Language Tools User's Guide. TEXAS Instruments. 2001
9 TMS320C6000 Optimizing Compiler User's Guide. TEXAS Instruments. 2001
10 TMS320C6201/6701 Evaluation Module User's Guide. TEXAS Instruments. 1998
11 TMS320C62x DSP Library Programmer's Reference. TEXAS Instruments. 2000
12 TMS320C6000 DSP/BIOS Application Programming Interface (API) Reference Guide. TEXAS Instruments . 2001
13 TMS320C54x Optimizing C/C++ Compiler User's Guide. TEXAS Instruments. 2001
14 ADSP2106X User's Guide. Analog Devices Inc. 1995
15 ADSP21160 SHARC Technical specifications. Rev.2.03. Analog Devices Inc. 1998
16 Code Composer Studio Getting Started Guide. TEXAS Instruments. 2001
17 TMS320 DSP/BIOS User's Guide. TEXAS Instruments. 2001
18 VisualDSP++ 3.0 C/C++ Compiler and Library Manual for SHARC DSPs Compiler. Analog Devices Inc. 2002
19 VisualDSP++ 2.0 User's Guide for ADSP－21xxx DSPs. Analog Devices Inc . 2001
20 VisualDSP++2.0 C/C++ Compiler & Library Manual for ADSP－21xxx DSPs. Analog Devices Inc. 2001
21 VisualDSP++2.0 Assembler and Preprocessor Manual for ADSP－21xxx DSPs. Analog Devices Inc. 2001
22 VisualDSP++2.0 Linker & Utilities Manual for ADSP－21xxx DSPs. Analog Devices Inc. 2001
23 VisualDSP++ Kernel (VDK) User's Guide. Analog Devices Inc . 2001
24 VisualDSP++ 2.0 Getting Started Guide for ADSP－21xxx DSPs. Analog Devices Inc. 2001
25 MATLAB® Link for Code Composer Studio™ Development Tools User's Guide, Mathworks., 2002
26 Embedded Target for the TI TMS320C6000™ DSP Platform User's Guide. Mathworks, 2002
27 Simulink Reference Mathworks， Version 5. Mathworks. 2002

28 Real - Time Workshop User's Guide，Version 5. Mathworks. 2002
29 任丽香，马淑芬，李方慧. TMS320C6000系列DSPs的原理与应用. 北京：电子工业出版社，2001
30 苏涛等. DSP实用技术. 西安：西安电子科技大学出版社，2002
31 苏涛，吴顺君，李真芳，宋万杰. 高性能DSP与高速实时信号处理(第二版). 西安：西安电子科技大学出版社，2002